Compilation and Language Theory

2nd edition

Stéphane GALLAND
Module LO46

IRTES-SeT, UTBM
90010 Belfort cedex, France
stephane.galland@utbm.fr – http://www.multiagent.fr
Goals of this Module

1. Study models, techniques and algorithms that permit to analyze a text-based language.

2. Study models, techniques and algorithms that permit to generate and execute code.

3. Study the techniques for the optimization of executable codes (available soon).
Outline

A  Overview of the Compilation Theory

B  Lexical Analysis

C  Syntax Analysis

D  Intermediate Code Generation

E  Run-time Environments
Languages:
- Java (tutorials and projects)
- C/C++ (projects)
- C# (projects)

Integrated Development Environment:
- Eclipse (tutorials and projects)
- NetBean (projects)
- Visual Studio (projects)

Compilation Tools:
- javacc (tutorials, projects)
- jlex (projects)
- lex, flex (projects)
- yacc, bison (projects)
Lectures in English.

- Supervised tutorials in English or French.
- Laboratory works mainly in French.
- Exams in English.
Evaluation of the Students

■ Project: 30% of the final score.
  ■ 2 to 4 students per group
  ■ subjects and guidelines will be detailed during the second lecture session.

■ Mid-term Exam: 25% of the final score.

■ Final Exam: 25% of the final score.

■ Laboratory Works: 20% of the final score.
  ■ Several practice sessions will be selected, and your works evaluated by teachers.
  ■ How many sessions? When? You will discover the answers at the beginning of each session.
To follow this module with the best results, you should have the following knowledge:

- You should be familiar with one of the following languages, and may have encountered other languages as well: C, C++, C#, or Java.

- You should have already experienced CLI compilation.

- You should have a good level in algorithmic.
Best Way to Follow the Lectures

1. Download the PDF files of the slides before the lecture.

2. Do not read each word of the slides during the lectures.

3. Listen carefully the teachers and takes notes on the side of the slides.


5. You must read the slides at home as soon as possible, not few hours before the exams.
Recommended Book

Compilers — Principles, Techniques and Tools Second Edition
2nd edition

Alfred V. AHO, Monica S. LAM, Ravi SETHI and Jeffrey D. ULLMAN

Pearson & Addison Wesley, 2007
ISBN 0-321-48681-1
Recommended Book

Parsing Techniques — A Practical Guide

Dick Grune and Ceriel J.H. Jacobs

Springer Verlag New York, 2007

Calculabilité, Complexité et Approximation

Jean-François REY

Vuibert, France, 2004

ISBN 2-7117-4808-1
Chapter 1
Overview of the Compilation Theory

Stéphane GALLAND
1 Introduction

2 Programming languages

3 What is a language processor?

4 Process of a compiler

5 Tools to create a compiler

6 Conclusion
Programming languages are notations for describing computations to people and to machines.

All the software running on all the computers was written in some programming language.

Before a program can be run, it first must be translated into a form in which it can be executed by a computer.

The software systems that do this translation are called compilers.

Goal of this chapter: give an overview of a typical simple compiler.

With this chapter you may understand the key points of language theory.
1 Introduction

2 Programming languages
   - Brief history
   - Classifications and types of programming languages
   - Basics of programming languages

3 What is a language processor?

4 Process of a compiler

5 Tools to create a compiler

6 Conclusion
1940’s: machine language, sequences of 0’s and 1’s.

1950’s: mnemonic assembly languages.

Later in 1950’s: Fortran for scientific computation, Cobol for business data processing, Lisp for symbolic computation. ALGOL (ALGOrithmic Language) is the ancestor of the modern languages such as B, Pascal, Simula and C.

1960’s and 1970’s: Refinements in the 3GL

- APL: array programming and functional programming
- PL/I: merging the best concepts from Fortan and Cobol.
- Simula: first OO language, followed by Smalltalk
- C: operating system programming
- Prolog: first logic programming language
- ML: polymorphic type system on top of Lisp
A Brief History of Programming Languages

Figure by Matthew Hancock — Complete figure on moodle.utbm.fr

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
Generations of Programming Languages

- **1st Generation – 1GL:** Machine languages.
- **2nd Generation – 2GL:** Assembly languages.
- **3rd Generation – 3GL:** High-level languages (Fortran, Cobol, Lisp, C, C++, C#, Java...)
- **4th Generation – 4GL:** languages designed for specific applications, like Nomad for report generation, SQL for database, Postscript for text formatting.
- **5th Generation – 5GL:** languages based on logic and constraints, like Prolog and OPS5.
Imperative or Declarative Language

**Imperative Languages**

- Imperative programs are specifying how a computation is to be done.
- Notion of program state and statements that change the state.
- **Examples:** C, C++, C#, Java.

**Declarative Languages**

- Declarative programs are specifying what computation is to be done.
- Functional and logic-based languages.
- **Examples:** ML, Haskell, Prolog.
Object-oriented Language

- Supports object-oriented programming.

- Consists in building programs from a collection of objects that interact with one another.

Examples

- Simula 67, Smalltalk (the earliest major OO languages)
- C++, C#, Java, Ruby....
Scripting Language

- Interpreted languages with high-level operators designed for “gluing together” computations.

- Programs are much shorter than equivalent program written in other languages.

Examples

Awk, Basic, JavaScript, Perl, PHP, Python, Ruby, Tcl, …
 Outline

1. Introduction
2. Programming languages
   - Brief history
   - Classifications and types of programming languages
   - Basics of programming languages
     - Definitions
     - Environment and state
     - Static or dynamic policy
     - Parameter-passing mechanisms
     - Aliasing mechanism
3. What is a language processor?
4. Process of a compiler
5. Tools to create a compiler
6. Conclusion
Definitions (#1)

- **Name**: a string of characters that refers to a thing in the program.

- **Identifier**: a string of characters that refers to an entity (data object, procedure, class, type).
  - All identifiers are names; but not all names are identifiers
  - x.y is a name but not an identifier, and x and y are identifiers.

- **Variable**: a particular location of the store of the values at run-time. A variable is denoted by a name. Each declaration of an identifier introduces a new variable.

- **Keyword**: an identifier that has a particular meaning to the programming language.
Definitions (#2)

- **Procedure**: a subprogram that may be called.
- **Function**: a procedure that may return a value of some type (the “return type”).
- **Method**: a procedure or a function inside a class in object-oriented languages.

**Caution**

In the C-family languages, all the subprograms are functions; and a function is enabled to return nothing (**void**).

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
Definitions (#3)

- **Declaration**: tells us about the type of a thing.
  - **Example**: `int i;`

- **Definition**: tells us about the value of a thing.
  - **Example**: `i = 1;`

- **Signature of a procedure**: the declaration of the procedure. It is composed of a return type, an identifier, and a collection of parameter declarations.

**Example**

In C++:

- a method is declared in a `.hpp` file.
- a method is defined in a `.cpp` file.
The association of names with locations in memory (the store) and then with values is described by two mappings:

- **Environment**: mapping from names to locations in the store.
- **State**: mapping from locations in store to their values.
Example of Environment and State

```c
... int i; /* global i */
...
void f(...) {
    int i; /* local i */
    ...
    i = 3; /* use of local i */
    ...
}
...

x = i + 1; /* use of global i */
```

Variable: i
- In: f
- Value: 3

Variable: i
- In: global scope
- Value: X

Name: i
One of the most important issues when designing a compiler is what decisions can the compiler make about the program.

**Static Policy**

A program uses a policy that enables the compiler to decide an issue; the decision could be decided at compile time.

**Dynamic Policy**

The decision can be made when we execute the program; the decision is required at run time.
A language uses a static scope if it is possible to determine the scope of a declaration by looking only at the program (C, Java…)

With dynamic scope, as the program runs, the same use of a variable x could refer to any of several different declarations of x (Perl, PHP…).
Here `static` refers not to the scope of the variable, but rather to the ability of the compiler to determine the location in memory.

If `static` is omitted each object has this variable and the compiler cannot determine where it is in advance.
The environment and state mappings are often dynamic.

**Static or Dynamic Environment Mapping?**
- Most of binding names to locations are dynamic.
- Some declarations (e.g., global i) are determined at compile time; they are static.

**Static or Dynamic State Mapping?**
- Most of binding locations to values are dynamic because it is impossible to determine the location until we run the program.
- Declared constants are an exception.
All programming languages have the notion of procedure.

But they can differ in how these procedures get their arguments.

How the actual parameters (the parameters used in the call of a procedure) are associated with the formal parameters (those used in the procedure definition)?

1. Call-by-value
2. Call-by-reference
3. Call-by-name
Passing Parameters with Call-by-value

- The actual parameter is evaluated or copied.
- The value is placed in the location belonging to the corresponding formal parameter of the called procedure.
- Used in C and Java; and the default option in C++.
- All computation involving the formal parameters done by the called procedure is local to that procedure; and
- The actual parameters themselves cannot be changed.

Caution

In Java, all the object variables are references, or pointers, to the objects. Parameters are passed with the call-by-value policy, not the call-by-reference.
The address of the actual parameter is passed to the callee as the value of the corresponding formal parameter.

Uses of the formal parameter in the code of the callee are implemented by following the pointer to the location indicated by the caller.

Changes to the formal parameter thus appear as changes to the actual parameter.
- It requires that the callee execute as if the actual parameter were substituted literally for the formal parameter in the code of the callee.

- Macro-functions in the C-family languages use this parameter passing mechanism.
Aliasing is occurring when two formal parameters refer to the same location.

Such variables are said to be aliases of one another.

It is possible when references to the objects are passed by value (as in Java).
Introduction
Programming languages
What is a language processor?
Process of a compiler
Tools to create a compiler
Conclusion
What is a Compiler?

- Read a program in one language — the source language.
- Translate it into an equivalent program in another language — the target language.
- Report any errors in the source program that are detected during the translation process.
Running the program

If the target program is an executable machine-language program, it can then be called by the user to process inputs and produce outputs.
What is an Interpreter?

- A kind of language processor.
- Does not produce a target program.
- Directly execute the operations specified in the source program on inputs supplied by the user.
What is an Hybrid Compiler?

- Combine compilation and interpretation.
- Generate intermediate program in a platform-independent language.
- Execute the intermediate program in a platform-dependent virtual machine.

Translator

(Compiler)

Source program

Intermediate program

Virtual Machine

(Processor)

Outputs

Inputs

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
Properties of the Language Processors

Compiler vs. Interpreter

- Compiler is faster than interpreter at mapping inputs to outputs.
- Interpreter gives better error diagnostics than compiler, because it executes the source program statement by statement (no code optimization).

Hybrid Compiler

- Hybrid compiler enables to compile on one machine, and to execute the generated program on another machine with a different low-level architecture than the initial machine, or across a network.
- In order to achieve faster processing, some hybrid compilers use just-in-time compilers to translate intermediate programs into machine language and avoid the interpretation, e.g., the Oracle’s Java Runtime Environment.
Several other programs may be required to create an executable target program.

They compose the toolchain of the compiler.
The Preprocessor in the Toolchain

Goals

- To collect the different files of the program’s modules to compile.
- To expand shorthands, macros into statements.
Goals

- To produce an assembly-language program from the modified source program.
- Assembly-language is easier to produce and debug.
Goals

- To translate to a machine code that could be relocated in the code segment of the program.
- Code segment: the part of the memory where machine code is stored
The Linker in the Toolchain

Goals

- To resolve external memory addresses, where the code in one file (library or object) may refer to a location in another file (library or object)
Outline

1. Introduction
2. Programming languages
3. What is a language processor?
4. Process of a compiler
5. Tools to create a compiler
6. Conclusion
The analysis breaks up the source program into constituent pieces and imposes a grammatical structure to them.

It detects if the source program is ill formed or semantically unsound.

It collects informations about the source program and stores it in a data structure called symbol table.

This part is often called the front end of the compiler.
Synthesis

- The synthesis constructs the desired target program from the intermediate representation and the information in the symbol table.
- This part is often called the back end of the compiler.
Lexical Analysis or Scanning

- The lexical analyzer reads the stream of characters making up the source program.
- It groups the characters into meaningful sequences called lexemes.
- For each lexeme, the lexical analyzer produces as output a token of the following form; and passes it to the syntactic analyzer.

  `<token-name, attribute-value>`

- The token-name is the identifier of the token.
- The attribute-value points to an entry in the symbol table for this token.
Example of Scanning

\[
\text{position} = \text{initial} + \text{rate} \times 60
\]

- position is a lexeme mapped into the token \(<\text{id},1>\); where id is an abstract symbol standing for identifier and “1” points to the symbol-table entry for position.
- \(=\) is a lexeme mapped into the token \(<\text{==}>\)
- initial is a lexeme mapped into the token \(<\text{id},2>\)
- \(+\) is a lexeme mapped into the token \(<\text{++}>\)
- rate is a lexeme mapped into the token \(<\text{id},3>\)
- \(*\) is a lexeme mapped into the token \(<\text{*}>\)
- 60 is a lexeme mapped into the token \(<\text{number},60>\)

© 2012–2014 Stéphane Galland  
stephane.galland@utbm.fr – http://www.multiagent.fr
Example of Scanning

position = initial + rate * 60

$id,1$$<=$id,2$$>$$+$$>$id,3$>$*<number,60>

Symbol Table

1 position float ...
2 initial float ...
3 rate float ...

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
Symbol Table

- The symbol table is a data structure containing a record for each variable name, with fields for the attributes of the name.

- The symbol table is filled or used by the different stages of the compiler.

- The attributes may provide information about the storage allocated for a name, its type, its scope, the number and types of the formal parameters, the method of passing each argument, and the type returned.

- The data structure should be designed to allow the compiler to find the record for each name quickly and to store or retrieve data from that record quickly.
Scopes are implemented by setting up a separate symbol table for each scope.

The most-closely nested rule for blocks permits to define a data structure, which is based on chained symbol tables.
Simple Java Implementation of the Symbol Table Chain (#1)

```java
/** Define the properties of a single symbol. */
public class Symbol {
    public final String lexeme;
    public Type type;
    public Address storagePosition;
    public Symbol(String lexeme) {
        this.lexeme = lexeme;
    }
}

/** Define a symbol table. */
public class SymbolTable {

    /** Collection of the symbol in the current context. */
    private final Map<String, Symbol> table = new TreeMap<String, Symbol>();

    /** Reference to the symbol table that is associated to the enclosing scope. */
    private final SymbolTable enclosingEnvironment;

    /** Constructor. */
    private SymbolTable(SymbolTable enclosingEnvironment) {
        this.enclosingEnvironment = enclosingEnvironment;
    }

    // ....
```
Simple Java Implementation of the Symbol Table Chain (#2)

```java
/** Declare a symbol in the current context. */
public void declare(String identifier, Symbol symbol) {
    this.table.put(identifier, symbol);
}

/** Get the definition of a symbol in the current context, or in an enclosing scope. */
public Symbol get(String identifier) {
    SymbolTable e = this;
    Symbol symbol;
    while (e != null) {
        symbol = e.table.get(identifier);
        if (symbol != null) {
            return symbol;
        }
        e = e.enclosingEnvironment;
    }
    return null;
}

// ....
```
/** Reference to the current symbol table. 
The reference is initialized with the 
root context (or the global context). */
private static SymbolTable current = new SymbolTable(null);

/** Replies the symbol table of the current context. */
public static SymbolTable getCurrent() {
    return current;
}

/** Open a new context and create the corresponding 
symbol table. */
public static void openContext() {
    current = new SymbolTable(current);
}

/** Close the current context. */
public static void closeContext() {
    if (current.enclosingEnvironment!=null) {
        current = current.enclosingEnvironment;
    }
}
The syntax analyzer uses the tokens produced by the lexical analyzer to create a tree-like intermediate representation.

A typical representation is a syntax tree:
- **node**: operation in the program
- **children**: parameters of the operation

In real-life compilers a context free grammar is used to describe the syntax of the language and to help to analyze this syntax.
Example of Syntax Analysis

position = initial + rate * 60

\[
\text{<id,1><}=><text{id,2}><+=><text{id,3}><*=><text{number,60}>}
\]

Symbol Table

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>position float ...</td>
</tr>
<tr>
<td>2</td>
<td>initial float ...</td>
</tr>
<tr>
<td>3</td>
<td>rate float ...</td>
</tr>
</tbody>
</table>

Character stream
Lexical Analyzer
Token stream
Syntax Analyzer
Syntax tree
Semantic Analyzer
Syntax tree
Intermediate Code Generator
Intermediate representation
Machine-Independent Code Optimizer
Intermediate representation
Code Generator
Target-machine code
Machine-Dependent Code Optimizer
Target-machine code

© 2012–2014 Stéphane Galland
stepphane.galland@utbm.fr – http://www.multiagent.fr
The semantic analyzer uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition.

It gathers type information and saves it in either the syntax tree and the symbol table.

Type checking is an important part of the semantic analyzer: the compiler checks that each operator has matching operands.

The semantic analyzer applies coercions, or type conversions.
Example of Semantic Analysis

position = initial + rate * 60

Symbol Table

1 position float ...
2 initial float ...
3 rate float ...

Character stream
Lexical Analyzer
Token stream
Syntax Analyzer
Syntax tree
Semantic Analyzer
Syntax tree
Intermediate Code Generator
Intermediate representation
Machine-Independent Code Optimizer
Intermediate representation
Machine-Dependent Code Optimizer
Target-machine code
Target-machine code

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
Many compilers generate an explicit low-level or machine-like intermediate representation, which is a program for an abstract machine.

The intermediate code generator should be:
- easy to produce, and
- easy to translate into the target machine.

Two intermediate representations are generally used:
- Syntax tree, and
- Three-address code
What is the Three-address Code?

- Three-address code consists of a sequence of assembly-like instructions with, at most, three operands per instruction.

  \[ \text{variable} = \text{operand1} \text{ operator } \text{operand2} \]

- Each operand can act like a register.

- The affectation operator is implicit and always present.

- **Constraints:**
  1. at most one operator on the right side.
  2. temporary names are generated to hold the value computed by the three-address instruction.
  3. some instructions have fewer then three operands.
Example of Intermediate Code Generation

position = initial + rate * 60

Symbol Table

1 position float ...
2 initial float ...
3 rate float ...

Intermediate representation

Character stream
Lexical Analyzer
Token stream
Syntax Analyzer
Syntax tree
Semantic Analyzer
Syntax tree
Intermediate Code Generator
Intermediate representation
Machine-Independent Code Optimizer
Intermediate representation
Code Generator
Target-machine code
Machine-Dependent Code Optimizer
Target-machine code
This code optimizer improves the intermediate code so that better target code will result.

“Better” means faster, shorter, less power consumer...

All the compilers include a machine-independent code optimizer.

But those that spent a large amount of time on this phase are named “optimizing compilers.”

Remember that many of the simple optimizations permit to significantly improve the running time of the target program without too much time spent on this phase.
Example of Optimization

```
t1=inttofloat(60)
t2=id3*t1
t3=id2+t2
id1=t3
```

Conversions of constants are replaced by the results of the conversions themselves

```
t1=60.0
```

Registers, when initialized with one operand on the right side, are replaced by the right side in the others instructions

```
t1=id3*60.0
id1=id2+t1
```
Example of Machine-Independent Code Optimization

\[
\text{position} = \text{initial} + \text{rate} \times 60
\]

Symbol Table

1. position float ...
2. initial float ...
3. rate float ...

Intermediate representation:

\[
\begin{align*}
\text{id1} &= \text{id2} + \text{id3} \\
\text{id3} &= \text{inttofloat}(60) \\
\text{id1} &= \text{id2} + \text{id3} \\
\text{id1} &= \text{id3} \times 60.0 \\
\end{align*}
\]
The code generator takes as input an intermediate representation and maps it into the target language.

If the target language is machine code, registers or memory locations are selected for each variables used by the program.

Then, the intermediate instructions are translated into sequences of machine instructions that perform the same tasks.

A crucial aspect is the judicious assignment of registers to hold variables.
 Assumes that R1 and R2 are registers.

 Variables are mapped to registers so that they can be easily used for the generation of the next instructions.
Example of Code Generation

position = initial + rate * 60

Symbol Table
1 position float ...
2 initial float ...
3 rate float ...

Example of Code Generation

Lexical Analyzer
Syntax Analyzer
Semantic Analyzer
Intermediate Code Generator
Machine-Independent Code Optimizer
Code Generator

Character stream
Token stream
Syntax tree
Syntax tree
Intermediate representation
Intermediate representation
Target-machine code
Target-machine code

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
Outline

1 Introduction
2 Programming languages
3 What is a language processor?
4 Process of a compiler
5 Tools to create a compiler
6 Conclusion
Several tools are available to help the compiler writer to build his compiler.

These tools use specialized languages for specifying and implementing specific components.

**Tools**

- **Parser generators**: produce syntax analyzers from grammatical description of a programming language (Yacc, JavaCC, Bison...)

- **Scanner generators**: produce lexical analyzers from a regular-expression description of the tokens of the languages (Flex, JFlex...)

- **Syntax-directed translation engines**: produce collections of routines for walking a parse tree and generating intermediate code.
Tools

- **Code-generator generators**: produce a code generator from a collection of rules for translating each operation of the intermediate language into the machine language for a target machine.

- **Data-flow analysis engines**: facilitate the gathering of information about how values are transmitted from one part of the program to each other part.

- **Compiler-construction toolkits**: provide an integrated set of routines for constructing various phases of a compiler.
Outline

1. Introduction
2. Programming languages
3. What is a language processor?
4. Process of a compiler
5. Tools to create a compiler
6. Conclusion
Key Concepts of the Chapter (#1)

- **Language Processors**: An integrated software development environment: compilers, interpreters, linkers, loaders, debuggers, profilers.

- **Compiler Phases**: Sequence of phases, each of which transforms the source program from one intermediate representation to another.

- **Machine and Assembly Languages**: Machine languages were the first-generation programming languages, followed by assembly languages.

- **Code Optimization**: the science of improving the efficiency of code in both complex and very important. It is a major portion of the study of compilation.
Higher-Level Languages: Programming languages take on progressively more of the tasks that formerly were left to the programmer: memory management, type-consistency...

Environments: The association of names with locations in memory and then with values can be described in terms of environments.

Parameter Passing: Parameters are passed from a calling procedure to the callee either by value or by reference.

Aliasing: When parameters are (effectively) passed by reference, two formal parameters can refer to the same object.
Compiler Front End: The part of the compiler that is dedicated to the analysis phases. The compiler front end takes the source program, breaks it to token, analyzes the grammar, detects errors and inconsistencies, and generate an intermediate representation.

Compiler Back End: The part of the compiler that is dedicated to the synthesis phases. The compiler back end takes the intermediate representation, generates assembly and machine code.

Lexical Analyzer: The lexical analyzer reads the input one character at a time and produces as output a stream of tokens. A token consists of a terminal symbol and attribute values.
Key Concepts of the Chapter (#4)

- **Parsing**: Parsing is the problem of figuring out how a string of terminals can be derived from the start symbol of the grammar by repeatedly replacing a nonterminal by the body of one of its productions.

- **Parse Tree**: A graphical tree representation of the productions that are matching a sequence of input tokens.

- **Intermediate Code**: The result of the syntax analysis is a representation of the source program, called intermediate code. Two primary forms of intermediate code are illustrated: abstract syntax tree (similar to parse tree), and three-address code.

- **Symbol Table**: A data structure that holds information about identifiers.
Bibliography of the Chapter (#1)

History of Programming Languages, volume 2 of ACM Press.
Addison Wesley.

Church, A.
The Calcul of Lambda Conversion.
Princeton University Press, Princeton, N.J.

Frege, G. (1967).
Begriffsschrift, a formula language, modeled upon that of arithmetic, for pure thought, chapter From Frege to Gödel.
Harvard Univ. Press, Cambridge, MA.

Programming Language Pragmatics.

Programming Languages: Concepts and Constructs.
Addison-Wesley.

History of Programming Languages, volume 1.
Academic Press.
Chapter 2
Lexical Analysis

Stéphane GALLAND
1 Introduction

2 Input buffering

3 Specification and recognition of tokens

4 Writing a lexical analyzer with Lex, Flex, JFlex, JavaCC

5 Writing a lexical analyzer by hand

6 Conclusion
1 Introduction
   - General principles
   - Definitions
   - Separating the lexical analyzer and the parser
   - Lexical errors

2 Input buffering

3 Specification and recognition of tokens

4 Writing a lexical analyzer with Lex, Flex, JFlex, JavaCC

5 Writing a lexical analyzer by hand
The lexical analyzer reads the stream of characters making up the source program.

- It groups the characters into meaningful sequences called lexemes.
- For each lexeme, the lexical analyzer produces as output a token of the following form; and passes it to the syntactic analyzer.
  
  \[ <\text{token-name, attribute-value}> \]

- The token-name is the identifier of the token.
- The attribute-value points to an entry in the symbol table for this token.
Lexical Analyzer (#2)

The tasks of the lexical analyzer of a compiler are:

1. discovering the tokens,
2. stripping the blanks and the comments,
3. correlating the error messages with the source program (line number tracking...)

Sometimes, lexical analyzers are divided into a cascade of two processes:

- **Scanning**: consists of the simple processes that do not require tokenization of the input: deletion of comments and compaction of consecutive white spaces.
- **Lexical analysis**: is the more complex portion, which produces tokens from the output of the scanner.
A lexeme is a sequence of characters in the source program that is identified by the lexical analyzer as a lexical unit (element of the language).

Example

- Let the statement: `printf("Total = %d\n", score);`
- Both `printf` and `score` are lexemes.
- The string of characters is a lexeme.
- The parenthesis, comma and semicolon characters are also lexemes.
A token is a pair consisting of a token name and an optional attribute value.

- The token name is an abstract symbol representing a kind of lexical unit.
- The token names are the input symbols that the parser processes.
- Generally the tokens are written in **boldface**.

**Example**

- Let the statement: `printf(”Total = %d\n”, score);`
- both `printf` and `score` are lexemes matching the pattern for token **id**.
A pattern is a description of the form that the lexemes of a token may take.

In the case of a keyword as a token, the pattern is just the sequence of characters that form the keyword.

For identifiers and some other tokens, the pattern is a more complex structure that is matched by many strings.

Example

Let the statement: `printf("Total = %d\n", score);`

both `printf` and `score` are described by the pattern `[_a-zA-Z][_a-zA-Z]*`
In many programming languages, the following classes cover most or all of the tokens:

1. One token for each keyword. The pattern is the name of the token itself;
2. Tokens for the operators, either individually or in classes such as the token `comparison`;
3. One token representing all identifiers;
4. One or more tokens representing constants, such as numbers and literal strings;
5. Tokens for each punctuation symbol, such as left and right parentheses, comma, and semicolon.
When more than one lexeme can match a pattern, the lexical analyzer must provide to the subsequent compiler phases additional information about the particular lexeme that matched. For example: the token `number` matches 0 and 1234.

The lexical analyzer returns to the parser the token name and an attribute value that describes the lexeme represented by the token.

We shall assume that tokens have at most one associated attribute. But it could be a data structure.

**Example**

The attribute value for the token `id` is an entry in the symbol table.
The lexical analyzer generally does not control the execution flow of the compiler.

The lexical analyzer is invoked by the parser through a call to `getNextToken`.

Then the lexical analyzer tries to discover and to reply a token.
There are number of reasons why the lexical analysis and the parsing are separated:

- **Simplicity of design**: The separation of lexical and syntactic analysis often allows us to simplify at least one of these tasks.

- **Compiler efficiency**: A separate lexical analyzer allows us to apply specialized techniques.

- **Compiler portability**: Input-device-specific peculiarities can be restricted to the lexical analyzer.
Lexical Errors

- It is hard for a lexical analyzer to tell that there is a source-code error.

- Let: \( \text{fi} (a == f(x)) \ldots \)

- The lexical analyzer cannot tell whether \( \text{fi} \) is a misspelling of the keyword \( \text{if} \) or an undeclared function identifier.

- The lexical analyzer fails when none of the patterns for tokens matches any prefix of the remaining input.

- If such an error is detected, the lexical analyzer must output an error message, and try to recover a stable state.
The simplest recovery strategy is the “panic mode.”

We delete successive characters from the remaining input, until the lexical analyzer can find a well-formed token at the beginning of what input is left.

Other possible error-recovery actions are:

- Delete one character from the remaining input.
- Insert a missing character into the remaining input.
- Replace a character by another character.
- Transpose two adjacent characters.
1 Introduction

2 Input buffering

3 Specification and recognition of tokens

4 Writing a lexical analyzer with Lex, Flex, JFlex, JavaCC

5 Writing a lexical analyzer by hand

6 Conclusion
The task of reading the source program is important and may be time consuming. It must be accelerated.

This task is made difficult by the fact that we often have to look one or more characters beyond the next lexeme before we can be sure we have the right lexeme.

For instance, we cannot be sure we have seen the end of an identifier until we see a character that is not a letter or a digit, and therefore is not part of the lexeme for `id`.

We shall introduce a two-buffer scheme that handles large lookaheads safely.
Specialized buffering techniques have been developed to reduce the amount of overhead required to process a single input character.

An important scheme involves two buffers that are alternatively reloaded:

Each buffer is of the same size $N$ (usually the size of the disk block).

Using one system read command, we can read $N$ characters into a buffer, rather than a call per character.
Using a Pair of Buffers (#2)

- A special character (**eof**) is put in the buffer when there is not enough characters in the input.

- Two pointers to the input are maintained:
  - Pointer **lexemeBegin**, marks the beginning of the current lexeme, whose extent we are attempting to determine.
  - Pointer **forward** scans ahead until a pattern match is found.
Once the next lexeme is determined, forward is set to the character at its right.

When the lexeme is recorded as an attribute value of a token, the lexemeBegin is set to the character immediately after the lexeme just found.

When forward is outside a buffer, the other buffer is reloaded from the input, and move forward to the beginning of the newly loaded buffer.

**Assumption**: the larger lexeme has a size lower or equals to $N$. 

\[ E = M \times C \times 2 \text{ eof} \]
In most modern languages, lexemes are short. Thus a buffer size $N$ in the thousands is ample, and the double-buffer scheme works without problem.

But some problems may occur: character strings can be very long, extending over many lines, then we could face the possibility that a buffer overflow occurs.

To avoid problems with long character strings, we can:
- add a dynamic buffer scheme for large lexeme; or
- reply to the parser a sequence of `str` tokens, one for each of the shorter strings.

Example

compile-time string concatenation in C: "ABC" "DEF"
According to the previously described double-buffer scheme, for each character read, we make two tests:

- one for the end of the buffer, and
- one to determine what character is read (usually a multiway branch).

To improve the speed of the treatment, we can combine the two tests by extending each buffer with a sentinel character (usually `eof`).
1 Introduction

2 Input buffering

3 Specification and recognition of tokens
   - Strings and operations on languages
   - Regular expressions
   - Regular definitions
   - Recognition of tokens

4 Writing a lexical analyzer with Lex, Flex, JFlex, JavaCC

5 Writing a lexical analyzer by hand
An alphabet is any finite set of symbols.  
**Example:** \( A = \{ a, b, c, \delta \} \)

A string over an alphabet is a finite sequence of symbols drawn from that alphabet.  
**Example:** \( s \in S = S(\mathcal{P}(A)) \setminus \{\emptyset\} = \{ a, b, c, \delta, ab, ac, a\delta, \ldots \} \)

The length of a string \( s \), usually written \( |s| \), is the number of occurrences of symbols in \( s \).

A language is any countable set of strings over some fixed alphabet.  
**Example:** \( L \subseteq S = \{ abc, \delta, b, bc \} \)
Operations on Languages (#1)

**Union of the languages $L$ and $M$**

$L \cup M = \{s | s \in L \lor s \in M\}$

**Example:** Let $L = \{a, b, c\}$ and $M = \{d, e\}$
then $L \cup M = \{a, b, c, d, e\}$

**Concatenation of the languages $L$ and $M$**

$LM = \{st | s \in L, t \in M\}$

**Example:** Let $L = \{a, b, c\}$ and $M = \{d, e\}$
then $L \cup M = \{ad, ae, bd, be, cd, ce\}$
Self-concatenation of the language $L$

$L^i = \begin{cases} 
\{\epsilon\} & \text{if } i = 0 \\
L^{i-1}L & \text{if } i > 0 
\end{cases}$

Example: Let $M = \{d, e\}$

then $M^4 = \{ddd, ddde, dded, ddee, dedd, dede, deded, deee, eddd, edde, edded, edee, eedd, eede, eeded, eeee\}$
Kleene’s Closure of the language $L$

$L^* = \bigcup_{i=0}^{\infty} L^i$

Example: Let $M = \{d, e\}$

then $M^* = \{\epsilon, d, e, dd, de, ed, ee, ddd, dde, ded, dee, edd, eed, eee, dddd, ddde, dded, ddee, dedd, dede, deed, deed, ...\}$

Positive Closure of the language $L$

$L^+ = \bigcup_{i=1}^{\infty} L^i$

Example: Let $M = \{d, e\}$

then $M^+ = \{d, e, dd, de, ed, ee, ddd, dde, ded, dee, edd, eed, eee, dddd, ddde, dded, ddee, dedd, dede, deed, deed, ...\}$
Outline

1 Introduction

2 Input buffering

3 Specification and recognition of tokens
   - Strings and operations on languages
   - Regular expressions
   - Regular definitions
   - Recognition of tokens

4 Writing a lexical analyzer with Lex, Flex, JFlex, JavaCC

5 Writing a lexical analyzer by hand
Regular expressions are commonly used for describing all the languages that can be built from the operators applied to the symbols of some alphabet.

The regular expressions are built recursively out of smaller regular expressions using the rules, which are described in the basis.

Each regular expression \( r \) denotes a language \( L(r) \), which is also defined recursively from the languages denoted by \( r \)'s expressions.
The rules that define the regular expressions over some alphabet $\Sigma$ and the languages that those expressions denote are:

**Definition (BASIS)**

There are two rules that form the basis:

1. $\epsilon$ is a regular expression, and $L(\epsilon)$ is $\{\epsilon\}$, that is, the language whose sole member is the empty string.

2. If $a$ is a symbol in $\Sigma$, then $a$ is a regular expression, and $L(a) = \{a\}$, that is, the language with one string, of length one, with $a$ in its position.

**Remark**

By convention, we use *italics* for symbols, and *boldface* for their corresponding regular expressions.
Definition (INDUCTION)

There are four parts to the induction whereby larger regular expressions are built from smaller ones. Suppose $r$ and $s$ are regular expressions denoting languages $L(r)$ and $L(s)$, respectively.

1. $(r)| (s)$ is a regular expression denoting the language $L(r) \cup L(s)$
2. $(r)(s)$ is a regular expression denoting the language $L(r)L(s)$
3. $(r)^*$ is a regular expression denoting the language $(L(r))^*$
4. $(r)$ is a regular expression denoting $L(r)$

This last rule says that we can add additional pairs of parentheses around expressions without changing the language they denote.
Regular expressions often contain unnecessary pairs of parentheses.

We may drop certain pairs if we adopt the convention that:

a) The unary operator "\*" has highest precedence and is left associative.

b) Concatenation has second highest precedence and is left associative.

c) "|" has lowest precedence and is left associative.
A language that can be defined by a regular expression is called a regular set.

If two regular expressions \( r \) and \( s \) denote the same regular set, we say they are equivalent and write \( r = s \).

<table>
<thead>
<tr>
<th>Law</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r</td>
<td>s = s</td>
</tr>
<tr>
<td>( r</td>
<td>(s</td>
</tr>
<tr>
<td>( r(st) = (rs)t )</td>
<td>Concatenation is associative</td>
</tr>
<tr>
<td>( r(s</td>
<td>t) = rs</td>
</tr>
<tr>
<td>( \epsilon r = r\epsilon = r )</td>
<td>( \epsilon ) is the identity for concatenation</td>
</tr>
<tr>
<td>( r^* = (r</td>
<td>\epsilon)^* )</td>
</tr>
<tr>
<td>( r^{**} = r^* )</td>
<td>( * ) is idempotent</td>
</tr>
</tbody>
</table>
Outline

1. Introduction

2. Input buffering

3. Specification and recognition of tokens
   - Strings and operations on languages
   - Regular expressions
   - Regular definitions
   - Recognition of tokens

4. Writing a lexical analyzer with Lex, Flex, JFlex, JavaCC

5. Writing a lexical analyzer by hand

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
Regular Definitions

- For notational convenience, we may wish to give names to certain regular expressions and use those names in subsequent expressions, as if the names were themselves symbols.
- If $\Sigma$ is an alphabet of basic symbols, then a regular definition is a sequence of definitions of the form:

$$d_1 \rightarrow r_1$$
$$d_2 \rightarrow r_2$$
$$\ldots$$
$$d_n \rightarrow r_n$$

- where each $d_i$ is a new symbol, not in and not the same as any other of the $d$’s, and
- each $r_i$ is a regular expression over the alphabet $\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$.
Well-known Regular Definitions

\begin{align*}
\text{letter} & \rightarrow A \mid B \mid \ldots \mid Z \mid a \mid b \mid \ldots \mid z \\
\text{letter}_- & \rightarrow \text{letter}_- \\
\text{digit} & \rightarrow 0 \mid 1 \mid \ldots \mid 9 \\
\text{letters} & \rightarrow \text{letter} \ \text{letter}^* \\
\text{digits} & \rightarrow \text{digit} \ \text{digit}^* \\
\text{id} & \rightarrow \text{letter}_-(\text{letter}_- \mid \text{digit})^* \\
\text{optFrac} & \rightarrow .\text{digits} \mid \epsilon \\
\text{optExp} & \rightarrow ((E \mid e)(+ \mid - \mid \epsilon)\text{digits}) \mid \epsilon \\
\text{number} & \rightarrow \text{digits} \ \text{optFrac} \ \text{optExp}
\end{align*}
Since Kleene introduced regular expressions with the basic operators in 1950s, many extensions have been added to enhance their ability to specify string patterns.

Most important notation add-ons:

1. **One or more instances**: The unary postfix operator "+" represents the positive closure of a regular expression and its language. \( L(r^+) = (L(r))^+ \). The operator has the same precedence and associativity as "*";

2. **Zero or one instance**: The unary operator "?" means “zero or one occurrence.” \( L(r?) = L(r) \cup \{\epsilon\} \). The operator has the same precedence and associativity as "*”;

3. **Character classes**: A regular expression \( a|b|\ldots|z \) can be replaced by \([ab\ldots z]\). If the symbols are consecutive, the expression could be written \([a−z]\).
Rewrite the Regular Definitions

\[
\begin{align*}
\text{letter} & \rightarrow [A - Za - Z] \\
\text{letter}_- & \rightarrow [A - Za - Z_] \\
\text{digit} & \rightarrow [0 - 9] \\
\text{letters} & \rightarrow \text{letter} + \\
\text{digits} & \rightarrow \text{digit} + \\
\text{id} & \rightarrow \text{letter}_-(\text{letter}_-|\text{digit})^* \\
\text{number} & \rightarrow \text{digits}(.\text{digits})?([Ee][+-]?\text{digits})? \\
\end{align*}
\]
Introduction

Input buffering

Token Recognition

Generators of lexical analyser

Lexical analyzer by hand

Conclusion

Outline

2 Input buffering

3 Specification and recognition of tokens

- Strings and operations on languages
- Regular expressions
- Regular definitions
- Recognition of tokens
  - Definition of the Lexeme-Token Pairs
  - Transition Diagram
  - Implementation of a Lexical Analyzer based on Transition Diagrams

4 Writing a lexical analyzer with Lex, Flex, JFlex, JavaCC

5 Writing a lexical analyzer by hand

© 2012–2014 Stéphane Galland

stephane.galland@utbm.fr – http://www.multiagent.fr
Now that we are able to express patterns using regular expressions, we must study how to take patterns for all the tokens of our language.

Consider the following language:

- statement → if expr then statement else statement
- expr → term = term, or
  expr → term <> term, or
  expr → term < term, or
  expr → term > term, or
  expr → term <= term, or
  expr → term >= term
- term → identifier, or
  term → number
## Definition of the Lexeme-Token Pairs (#2)

<table>
<thead>
<tr>
<th>Lexeme</th>
<th>Regular Expression</th>
<th>Token</th>
<th>Token Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ws</code></td>
<td><code>[ \n\t\r]</code>+</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><code>if</code></td>
<td><code>if</code></td>
<td><code>if</code></td>
<td>-</td>
</tr>
<tr>
<td><code>then</code></td>
<td><code>then</code></td>
<td><code>then</code></td>
<td>-</td>
</tr>
<tr>
<td><code>else</code></td>
<td><code>else</code></td>
<td><code>else</code></td>
<td>-</td>
</tr>
<tr>
<td><code>id</code></td>
<td>`letter_ (letter_</td>
<td>digit)∗`</td>
<td><code>id</code></td>
</tr>
<tr>
<td><code>number</code></td>
<td><code>digits(.digits)?</code></td>
<td><code>number</code></td>
<td>pointer to symbol table’s entry</td>
</tr>
<tr>
<td><code>=</code></td>
<td><code>=</code></td>
<td><code>relop</code></td>
<td><code>EQ</code></td>
</tr>
<tr>
<td><code>&lt;&gt;</code></td>
<td><code>&lt;&gt;</code></td>
<td><code>relop</code></td>
<td><code>NE</code></td>
</tr>
<tr>
<td><code>&lt;</code></td>
<td><code>&lt;</code></td>
<td><code>relop</code></td>
<td><code>LT</code></td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td><code>&gt;</code></td>
<td><code>relop</code></td>
<td><code>GT</code></td>
</tr>
<tr>
<td><code>&lt;=</code></td>
<td><code>&lt;=</code></td>
<td><code>relop</code></td>
<td><code>LE</code></td>
</tr>
<tr>
<td><code>&gt;=</code></td>
<td><code>&gt;=</code></td>
<td><code>relop</code></td>
<td><code>GE</code></td>
</tr>
</tbody>
</table>
2 Input buffering

3 Specification and recognition of tokens
   ■ Strings and operations on languages
   ■ Regular expressions
   ■ Regular definitions
   ■ Recognition of tokens
      ■ Definition of the Lexeme-Token Pairs
      ■ Transition Diagram
      ■ Implementation of a Lexical Analyzer based on Transition Diagrams

4 Writing a lexical analyzer with Lex, Flex, JFlex, JavaCC

5 Writing a lexical analyzer by hand
As an intermediate step in the construction of a lexical analyzer, we first convert patterns into flowcharts, called **transition diagrams**.

**Definition (Transition Diagram)**

- **Diagram**: is composed of states and edges.
- **State**: a step in the scanning of a string, that also indicates if the input stream is validating the regular expression, or not.
- **Edge**: directed from one state to another. Each edge is labeled by a symbol or a set of symbols.

**Assumption**

All transition diagrams are deterministic: never more than one edge out of a given state with a given symbol among its labels.
Example of a Transition Diagram for relop

```
start

0

1 < 2

1 = 2

1 > 3

3 <relop,LE>

3 <relop,NE>

3 <relop,LT>

3 <relop,EQ>

3 <relop,GE>

3 <relop,GT>

other

5

6 = 7

6 > 8

8 <relp,GT>
```

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
### Specific Notations

<table>
<thead>
<tr>
<th>Notation</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>start</strong></td>
<td>The transition diagram always begins in the start state before any input symbols have been read.</td>
</tr>
<tr>
<td>0 (\rightarrow) 1</td>
<td>The transition labelled with other is traversable when no other transition is traversable.</td>
</tr>
<tr>
<td>2</td>
<td>The accepting state (or final) indicates that a lexeme has been found (between pointers lexemeBegin and forward).</td>
</tr>
<tr>
<td>4 *</td>
<td>If the lexeme does not include the symbol that got us to the accepting state, it is necessary to retract the forward pointer by one position.</td>
</tr>
</tbody>
</table>
The transition diagram that recognizes the identifiers is:

```
9 11
start letter other *
letter or digit
10
<id, lexeme()>
```

- lexeme() replies the current lexeme (between lexemeBegin and forward pointers).

Recognizing keywords and identifiers presents a specific problem: keywords are not identifiers even though they look like identifiers.
There are two ways that we can handle reserved words that look like identifiers:

1. Install the keywords in the symbol table initially. A field of the symbol-table entry indicates that these strings are never ordinary identifiers.

   - `installID()` places the identifier in the symbol table if it is not already there and returns a pointer to the symbol-table entry.
   - `getToken()` replies the token that is corresponding to the lexeme, or `id` otherwise.
Create a separate transition diagram for each keyword.

- If we adopt this approach, then we must prioritize the tokens so that the reserved-word tokens are recognized in preference to id, when the lexeme matches both patterns.
- This approach is less used than the previous approach when the lexical analyzer is written by hand.
Outline

2 Input buffering

3 Specification and recognition of tokens
   ■ Strings and operations on languages
   ■ Regular expressions
   ■ Regular definitions
   ■ Recognition of tokens
     ■ Definition of the Lexeme-Token Pairs
     ■ Transition Diagram
     ■ Implementation of a Lexical Analyzer based on Transition Diagrams

4 Writing a lexical analyzer with Lex, Flex, JFlex, JavaCC

5 Writing a lexical analyzer by hand
There are several ways that a collection of transition diagrams can be used to build a lexical analyzer.

We may assume a variable state holding the number of the current state for a transition diagram.

Each transition diagram is simulated by a piece of code inside a function.

The code of a state is itself a switch statement or a multiway branch that determines the next state by reading and examining the next input character.
Example of Code for the Token relop

```java
Token getRelop() {
    /* return null on failure */
    char c;
    Token token = new Token(Tag.RELOP);
    while (true) {
        /* repeat until a return or failure */
        switch(state) {
            case 0:
                c = nextChar();
                if (c=='<') state = 1;
                else if (c=='=' ) state = 5;
                else if (c=='>') state = 6;
                else return null; /* lexeme is not a relop */
                break;
            case 1: ...
            case 8:
                retract();
                token.attribute = "GT";
                return token;
            default: return null;
        }
    }
}
```
To build the entire lexical analyzer, the codes for simulating the transition diagrams may be arranged in different ways.

1. We could arrange for the transition diagrams for each token to be tried sequentially.
   - Then when the function is replying null (failure), the pointer forward is reset and the next transition diagram is started.
   - This approach allows us to use the transition diagrams for the individual keywords.
   - We have only to use these before we use the diagram for id, in order the keywords to be reserved words.
We could run the various transition diagrams “in parallel”.

- If we use this strategy, we must be careful to resolve the case where one diagram finds a lexeme that matches its pattern, while one or more other diagrams are still able to process input.
- The normal strategy is to take the longest prefix of the input that matches any pattern.
The preferred approach is to combine all the transition diagrams into one.

- We allow the transition diagram to read input until there is no possible next state, and then take the longest lexeme that matched any pattern.
- The problem of combining transition diagrams for several tokens is complex. The easiest way to solve this problem is to study how lexical-analyzer generators, such as Lex or Flex, are working.
1 Introduction

2 Input buffering

3 Specification and recognition of tokens

4 Writing a lexical analyzer with Lex, Flex, JFlex, JavaCC
   - Generators of Lexical Analyser
   - Use of Lex
   - Lex program
   - Java generators

5 Writing a lexical analyzer by hand
Several tools allow to generate a lexical analyzer by specifying regular expressions to describe the patterns for the tokens.

This section introduces the tool Lex, and its more recent implementation Flex (dedicated to compilers written in C or C++).

The input notation is the Lex language.
Outline

1. Introduction
2. Input buffering
3. Specification and recognition of tokens
4. Writing a lexical analyzer with Lex, Flex, JFlex, JavaCC
   - Generators of Lexical Analyser
   - Use of Lex
   - Lex program
   - Java generators
5. Writing a lexical analyzer by hand
Introduction
Input buffering
Token Recognition
Generators of lexical analyser
Lexical analyzer by hand
Conclusion

Process of Lex

Lex Program: file.l
Lex Compiler
C Program: file.yy.c

C Program: file.yy.c
C Compiler
C Program: main.c
Executable: a.out

Input Stream
a.out
Sequence of tokens

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
The normal use of the program generated by the Lex compiler is as a subroutine of the parser.

The lexical analyzer is a C function that returns an integer, which is a code for one of the possible token names.

The attribute value, whether it another numeric code, a pointer to the symbol table, or nothing, is placed in a global variable yylval (shared between the lexical analyzer and the parser).
Outline

1. Introduction
2. Input buffering
3. Specification and recognition of tokens
4. Writing a lexical analyzer with Lex, Flex, JFlex, JavaCC
   - Generators of Lexical Analyser
   - Use of Lex
   - Lex program
   - Java generators
5. Writing a lexical analyzer by hand
A Lex program has the following form:

Declarations

manifest constants: identifiers declared to stand for a constant, eg. the name of a token,

regular definitions.
A Lex program has the following form:

- **Declarations**
  - `%%`
  - **Translation rules**
  - `%%`
  - **Auxiliary functions**

**Translation rules**

- have the form:

  ```
  Pattern { Action }
  ```

- Each pattern is a regular expression, which may use the regular definitions of the declaration section.
- The actions are fragments of code, usually in C.
- The first rule, which is matching, is used.
A Lex program has the following form:

```
Editions
%%
Translation rules
%%
Auxiliary functions
```

Auxiliary functions

- holds whatever additional functions are used in the actions.
Example of a Lex Program: declarations

```c
/* definitions of manifest constants, if not already declared in the parser files (yacc) */
enum { LT, LE, EQ, NE, GT, GE } RelopId;
enum { IF, THEN, ELSE, ID, NUMBER, RELOP } TokenName;

/* regular expressions */
delim \s+\n
ws { delim }+
letter [A-Za-z]
digit [0-9]
id { letter }({ letter }|{ digit })*
number { digit }+(\.{ digit }+)?([Ee][+-]?{ digit }+)?
```
Example of a Lex Program: translation rules

```c
{ws} { /* no action and no return */ }
if { return IF; }
then { return THEN; }
else { return ELSE; }
{id} { yylval = (int)installID(); return ID; }
{number} { yylval = (int)installNumber(); return NUMBER; }
"<" { yylval = LT; return RELOP; }
"<=" { yylval = LE; return RELOP; }
"==" { yylval = EQ; return RELOP; }
"<>" { yylval = NE; return RELOP; }
">" { yylval = GT; return RELOP; }
">=" { yylval = GE; return RELOP; }
```
Example of a Lex Program: auxiliary functions

```c
int installID() {
    /* function to install the lexeme, whose first character is pointed to by yytext, and whose length is yyleng, into the symbol table and return a pointer thereto */
}

int installNumber() {
    /* similar to installID, but puts numerical. Constants into a separate table */
}
```
Two rules are used by Lex to decide on the proper lexeme to select, when several prefixes of the input match one or more patterns:

1. Always prefer a longer prefix to a shorter prefix.

2. If the longest possible prefix matches two or more patterns, prefer the pattern listed first in the Lex program.
Lookahead Operator

- Lex automatically reads one character ahead of the last character that forms the selected lexeme, and then retracts the input so only the lexeme itself is consumed from the input.

- **Problem:** Sometimes, we want a certain pattern to be matched to the input only when it is followed by a certain other characters.

- **Solution:** use the character "/" in the pattern to indicate the end of the part of the pattern that matches the lexeme.
  - a / b means “a followed by b” (a and b are regular expressions)
  - The additional pattern (b) is not consumed from the input in the lexical analyzer point-of-view.
Outline

1. Introduction
2. Input buffering
3. Specification and recognition of tokens
4. Writing a lexical analyzer with Lex, Flex, JFlex, JavaCC
   - Generators of Lexical Analyser
   - Use of Lex
   - Lex program
   - Java generators
5. Writing a lexical analyzer by hand

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
Several implementations of lexical-analyzer generators provides Java source code.

JLex is a lexical analyzer generator, written for Java, in Java.

JLex is based upon the Lex lexical analyzer generator model ⇒ the input file is the similar as the one for Lex, but not the same.

User code
%%
JLex directives
%%
Translation rules

JLex Program

- **User code:** copied verbatim into the lexical analyzer source file.

- **JLex directives:** explained in the online documentation.

- **Translation rules:** series of rules for breaking the input stream into tokens.

  Each rule has three distinct parts: the optional state list, the regular expression, and the associated action:

  
  ```
  [<states>] <expression> { <action> }
  ```

```user code
% %
JLex directives
% %
Translation rules
```
JFLex

- JFLex is a lexical analyzer generator, written for Java, in Java.

- It is a rewrite of JLex with extended features (as for Flex/Lex implementations).

http://www.jflex.de
Java Compiler Compiler (JavaCC) is the most popular parser generator for use with Java applications.

Even if JavaCC is a parser, it includes a lexical analyzer in a transparent way.

The lexical specifications such as regular expressions, strings, etc. and the grammar specifications (the BNF) are both written together in the same file.

JavaCC is detailed in Chapter 194.

http://javacc.java.net
Outline

1. Introduction
2. Input buffering
3. Specification and recognition of tokens
4. Writing a lexical analyzer with Lex, Flex, JFlex, JavaCC
5. Writing a lexical analyzer by hand
   - Finite automata
   - Building a Lexical Analyzer
6. Conclusion
To go deeper in how a program like Lex turns its input program into a lexical analyzer, the formalism called “finite automata” is at the heart of this transition.
Finite automata are recognizers: they say “yes” or “no” about each possible input string.

Finite automata come in two flavors:

- **Nondeterministic finite automata**: NFA have no restrictions on the labels of their edges.
- **Deterministic finite automata**: DFA have, for each state, and for each symbol of its input alphabet exactly one edge with that symbol leaving that state.

NFA and DFA are represented by transition graphes.

Similar to transition diagram, except the same label can label edges from one state, and an edge may be labeled by $\epsilon$. 
1 Introduction

2 Input buffering

3 Specification and recognition of tokens

4 Writing a lexical analyzer with Lex, Flex, JFlex, JavaCC

5 Writing a lexical analyzer by hand
   - Finite automata
     - Nondeterministic finite automata
     - Deterministic finite automata
     - From regular expression to NFA
     - From NFA to DFA

Building a Lexical Analyzer
A nondeterministic finite automata (NFA) consists of:

1. A finite set of states $S$.

2. A set of input symbols $\Sigma$, the input alphabet. We assumed that $\epsilon$ is never a member of $\Sigma$.

3. A transition function that gives, for each state, and for each symbol in $\Sigma \cup \{\epsilon\}$ a set of next states.

4. A state $s_0$ from $S$ that is the start state or initial state.

5. A set of states $F \subseteq S$, that are the accepting states or final states.
The regular expression "\((a|b)^* \text{abb}\)" is described by the following NFA:
An NFA accepts input strings $x$ iff there is some path, such that the symbols along the path spell out $x$.

Note that $\epsilon$ labels are ignored in the path.

The language accepted by an NFA is the set of strings that have a path in the NFA.
Algorithm for Simulating a NFA

Input : An input string \( x \) terminated by the `eof` character. A NFA \( N \) with start state \( s_0 \), accepting states \( F \), and transition function \( \text{move} \).

Output : Answer "yes" if \( N \) accepts \( x \); "no" otherwise.

Method : The algorithm keeps a set of current states \( S \), those that are reached from \( s_0 \) following a path labeled by the inputs read so far. If \( c \) is the next input character, read by the function \( \text{nextChar} \), then we first compute \( \text{move} (S,c) \) and then close that set using \( \epsilon \)-closure.

\[
\text{begin} \\
\hspace{1cm} S \leftarrow \epsilon\text{-closure} \left( s_0 \right); \\
\hspace{1cm} c \leftarrow \text{nextChar}; \\
\hspace{1cm} \textbf{while} \; c \neq \text{eof} \; \textbf{do} \\
\hspace{2cm} S \leftarrow \epsilon\text{-closure} \left( \text{move} \left( S, c \right) \right); \\
\hspace{2cm} c \leftarrow \text{nextChar}; \\
\hspace{1cm} \textbf{end} \\
\text{return} \; S \cap F \neq \emptyset; \\
\text{end}
\]
## Functions in the Algorithm

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\varepsilon$-closure($s$)</td>
<td>Set of NFA states reachable from NFA state $s$ on $\varepsilon$-transitions alone.</td>
</tr>
<tr>
<td>$\varepsilon$-closure($T$)</td>
<td>Set of NFA states reachable from some NFA state $s$ in set $T$ on $\varepsilon$-transitions alone.</td>
</tr>
<tr>
<td>move($T, a$)</td>
<td>Set of NFA states to which there is a transition on input symbol $a$ from some state $s$ in $T$.</td>
</tr>
</tbody>
</table>
Example of NFA Simulation

Let the input: ”abababb”

```
begin
  S ← ε-closure (s₀);
  c ← nextChar;
  while c ≠ eof do
    S ← ε-closure (move (S, c));
    c ← nextChar;
  end
return S ∩ F ≠ ∅;
end
```

Example

S = {0}
forward: abababb
Example of NFA Simulation

Let the input: "abababb"

begin
    S ← ε-closure (s₀);
    c ← nextChar;
    while c ≠ eof do
        S ← ε-closure (move (S,c));
        c ← nextChar;
    end
    return S ∩ F ≠ ∅;
end

Example

S = {0}
c = a
forward: bababb
Example of NFA Simulation

Let the input: "abababb"

\[\begin{align*}
\text{let } S &\leftarrow \epsilon\text{-closure } (s_0); \\
\text{let } c &\leftarrow \text{nextChar}; \\
\text{while } c \neq \text{eof} \text{ do } \\
\quad &\text{let } S \leftarrow \epsilon\text{-closure } (\text{move } (S, c)); \\
\quad &\text{let } c \leftarrow \text{nextChar}; \\
\text{end} \\
\text{return } S \cap F \neq \emptyset;
\end{align*}\]

Example

\[\begin{align*}
S &\leftarrow \{0\} \\
c &\leftarrow a \\
\text{move}(\{0\}, a) &\leftarrow \{0, 1\} \\
\epsilon\text{-closure}(\{0, 1\}) &\leftarrow \{0, 1\} \\
S' &\leftarrow \{0, 1\}
\end{align*}\]
Example of NFA Simulation

Let the input: "abababb"

\[
\begin{align*}
S & \leftarrow \epsilon\text{-closure}(s_0); \\
c & \leftarrow \text{nextChar}; \\
\text{while } c \neq \text{eof} \text{ do} & \\
    S & \leftarrow \epsilon\text{-closure}(\text{move}(S,c)); \\
    c & \leftarrow \text{nextChar}; \\
\text{end} \\
\text{return } S \cap F \neq \emptyset;
\end{align*}
\]

Example

\[S = \{0, 1\}\]
\[c = b\]
forward: ababb
Example of NFA Simulation

Let the input: "abababb"

\[
\begin{align*}
\text{begin} & \quad S \leftarrow \epsilon\text{-closure} (s_0); \\
& \quad c \leftarrow \text{nextChar}; \\
& \text{while } c \neq \text{eof} \text{ do} \\
& \quad \quad S \leftarrow \epsilon\text{-closure} (\text{move} (S,c)); \\
& \quad \quad c \leftarrow \text{nextChar}; \\
\text{end} & \quad \text{return } S \cap F \neq \emptyset;
\end{align*}
\]

Example

\[
S = \{0, 1\} \\
c = b \\
\text{move}(\{0, 1\}, b) = \{0, 2\} \\
\epsilon\text{-closure}(\{0, 2\}) = \{0, 2\} \\
S' = \{0, 2\}
\]
Example of NFA Simulation

Let the input: "abababb"

```
begin
  S ← ε-closure (s0);
  c ← nextChar;
  while c ≠ eof do
    S ← ε-closure (move (S,c));
    c ← nextChar;
  end
return S ∩ F ≠ ∅;
end
```

Example

S = {0, 2}
c = a
forward: babb
Example of NFA Simulation

Let the input: ”abababb”

```
begin
    S ← \(\varepsilon\)-closure \(s_0\);
    c ← nextChar;
    while \(c \neq \text{eof}\) do
        S ← \(\varepsilon\)-closure \(\text{move}(S, c)\);
        c ← nextChar;
    end
    return \(S \cap F \neq \emptyset\);
end
```

Example

\(S = \{0, 2\}\)
\(c = a\)
move(\(\{0, 2\}, a\)) = \{0\}
\(\varepsilon\)-closure(\(\{0\}\)) = \{0\}
\(S' = \{0\}\)
Example of NFA Simulation

Let the input: ”abababb”

![NFA diagram]

```
begin
    S ← ϵ-closure (s₀);
    c ← nextChar;
    while c ≠ eof do
        S ← ϵ-closure (move (S,c));
        c ← nextChar;
    end
    return S ∩ F ≠ ∅;
end
```

Example

- $S = \{0\}$
- $c = b$
- forward: abb
Example of NFA Simulation

Let the input: "abababb"

```
begin
    S ← ϵ-closure (s0);
    c ← nextChar;
    while c ≠ eof do
        S ← ϵ-closure (move (S, c));
        c ← nextChar;
    end
    return S ∩ F ≠ ∅;
end
```

**Example**

- \( S = \{0\} \)
- \( c = b \)
- \( \text{move}(\{0\}, b) = \{0\} \)
- \( \epsilon\text{-closure}(\{0\}) = \{0\} \)
- \( S' = \{0\} \)
Let the input: ”abababb”

Example of NFA Simulation

begin
S ← ε-closure (s₀);
c ← nextChar;
while c ≠ eof do
    S ← ε-closure (move (S,c));
    c ← nextChar;
end
return S ∩ F ≠ ∅;
end

Example
S = {0}
c = a
forward: bb
Example of NFA Simulation

Let the input: ”abababb”

\[
\begin{align*}
\text{begin} & \quad S \leftarrow \varepsilon\text{-closure}(s_0); \\
\text{c} & \leftarrow \text{nextChar}; \\
\text{while } c \neq \text{eof} \text{ do} & \quad S \leftarrow \varepsilon\text{-closure}(\text{move}(S,c)); \\
\text{c} & \leftarrow \text{nextChar}; \\
\text{end} & \quad \text{return } S \cap F \neq \emptyset;
\end{align*}
\]

Example

\[
\begin{align*}
S & = \{0\} \\
c & = a \\
\text{move}(\{0\}, a) & = \{0, 1\} \\
\varepsilon\text{-closure}(\{0, 1\}) & = \{0, 1\} \\
S' & = \{0, 1\}
\end{align*}
\]
Example of NFA Simulation

Let the input: ”abababb”

begin
    \( S \leftarrow \varepsilon\text{-closure}(s_0); \)
    \( c \leftarrow \text{nextChar}; \)
    while \( c \neq \text{eof} \) do
        \( S \leftarrow \varepsilon\text{-closure}(\text{move}(S,c)); \)
        \( c \leftarrow \text{nextChar}; \)
    end
    return \( S \cap F \neq \emptyset; \)
end

Example
\( S = \{0, 1\} \)
\( c = b \)
foward: b
Example of NFA Simulation

Let the input: ”abababb”

![Diagram of NFA Simulation](image)

begin
    \( S \leftarrow \varepsilon\text{-closure}(s_0); \)
    \( c \leftarrow \text{nextChar}; \)
    while \( c \neq \text{eof} \) do
        \( S \leftarrow \varepsilon\text{-closure}(\text{move}(S,c)); \)
        \( c \leftarrow \text{nextChar}; \)
    end
    return \( S \cap F \neq \emptyset; \)
end

Example

\( S = \{0, 1\} \)
\( c = b \)
\( \text{move}(\{0, 1\}, b) = \{0, 2\} \)
\( \varepsilon\text{-closure}(\{0, 2\}) = \{0, 2\} \)
\( S' = \{0, 2\} \)
Example of NFA Simulation

Let the input: ”abababb”

```
begin
   S ← ε-closure (s0);
   c ← nextChar;
   while c ≠ eof do
      S ← ε-closure (move (S,c));
      c ← nextChar;
   end
   return S ∩ F ≠ ∅;
end
```

Example

$S = \{0, 2\}$

$c = b$

foward: eof
Example of NFA Simulation

Let the input: "abababb"

begin
  \( S \leftarrow \varepsilon\text{-closure}(s_0); \)
  \( c \leftarrow \text{nextChar}; \)
  while \( c \neq \text{eof} \) do
    \( S \leftarrow \varepsilon\text{-closure}(\text{move}(S,c)); \)
    \( c \leftarrow \text{nextChar}; \)
  end
  return \( S \cap F \neq \emptyset; \)
end

Example

\( S = \{0, 2\} \)
\( c = b \)
\( \text{move}(\{0, 2\}, b) = \{0, 3\} \)
\( \varepsilon\text{-closure}(\{0, 3\}) = \{0, 3\} \)
\( S' = \{0, 3\} \)
Example of NFA Simulation

Let the input: ”abababb”

\[
\begin{align*}
S & \leftarrow \epsilon\text{-closure } (s_0); \\
\text{c} & \leftarrow \text{nextChar}; \\
\text{while } \text{c} \neq \text{eof do} & \\
| & \quad S \leftarrow \epsilon\text{-closure } (\text{move } (S,c)); \\
| & \quad \text{c} \leftarrow \text{nextChar}; \\
\text{end} & \\
\text{return } S \cap F \neq \emptyset;
\end{align*}
\]

Example

\[S = \{0, 3\}\]
\[c = \text{eof}\]
Example of NFA Simulation

Let the input: "abababb"

\[
\begin{align*}
\text{begin} & \quad S \leftarrow \varepsilon\text{-closure}(s_0); \\
& \quad c \leftarrow \text{nextChar}; \\
& \quad \textbf{while } c \neq \text{eof} \textbf{ do} \\
& \quad \quad S \leftarrow \varepsilon\text{-closure}(\text{move}(S, c)); \\
& \quad \quad c \leftarrow \text{nextChar}; \\
\text{end} & \quad \text{return } S \cap F \neq \emptyset; \\
\text{end}
\end{align*}
\]

Example

\[
S = \{0, 2\} \\
F = \{3\} \\
S \cap F = \{0, 3\} \cap \{3\} = \{3\}
\]

Return "true"
1 Introduction

2 Input buffering

3 Specification and recognition of tokens

4 Writing a lexical analyzer with Lex, Flex, JFlex, JavaCC

5 Writing a lexical analyzer by hand
   - Finite automata
     - Nondeterministic finite automata
     - Deterministic finite automata
     - From regular expression to NFA
     - From NFA to DFA
     - Building a Lexical Analyzer
A deterministic finite automata (DFA) is a special case of an NFA where:

1. There are no moves on input $\epsilon$, and
2. For each state $s$ and input symbol $a$, there is exactly one edge out of $s$ labeled $a$.

While the NFA is used to recognize the strings of a language, the DFA is a simple and concrete algorithm for recognizing strings.

Every regular expression and every NFA can be converted to a DFA accepting the same language.

The lexical analyzers are built upon DFA.
The regular expression \((a|b)^* abb\) is described by the following DFA:
Algorithm for Simulating a DFA

Input : An input string $x$ terminated by $\texttt{eof}$ character. A DFA $D$ with start state $s_0$, accepting states $F$, and transition function $\text{move}$.
Output : Answer “yes” if $D$ accepts $x$; “no” otherwise.
Method : Apply algorithm on $x$. The function $\text{move}(s,c)$ gives the state to which there is an edge from state $s$ on input $c$. The function $\text{nextChar}$ returns the next character of the input string $x$.

begin
    $s \leftarrow s_0$;
    $c \leftarrow \text{nextChar}$;
    while $c \neq \texttt{eof}$ do
        $s \leftarrow \text{move}(s,c)$;
        $c \leftarrow \text{nextChar}$;
    end
    return $s \in F$;
end
Example of DFA Simulation

Let the input: "abababb"

begin
s ← s0;
c ← nextChar;
while c ≠ eof do
    s ← move (s, c);
    c ← nextChar;
end
return s ∈ F;
end

Example
s = 0
foward: abababb
Example of DFA Simulation

Let the input: "abababb"

begin
s ← s0;
c ← nextChar;
while c ≠ eof do
    s ← move (s, c);
c ← nextChar;
end
return s ∈ F;
end

Example
s = 0
c = a
forward: bababb
Example of DFA Simulation

Let the input: "abababb"

```
begin
    s ← s_0;
    c ← nextChar;
    while c ≠ eof do
        s ← move (s, c);
        c ← nextChar;
    end
    return s ∈ F;
end
```

**Example**

- \( s = 0 \)
- \( c = a \)
- \( \text{move}(0, a) = 1 \)
- \( s' = 1 \)
Example of DFA Simulation

Let the input: "abababb"

begin
    s ← s₀;
    c ← nextChar;
    while c ≠ eof do
        s ← move (s, c);
        c ← nextChar;
    end
    return s ∈ F;
end

Example

s = 1
c = b
forward: ababb
Example of DFA Simulation

Let the input: "abababb"

```
begin
    s ← s0;
    c ← nextChar;
    while c ≠ eof do
        s ← move (s, c);
        c ← nextChar;
    end
    return s ∈ F;
end
```

Example

\[
\begin{align*}
    s &= 1 \\
    c &= b \\
    \text{move}(1, b) &= 2 \\
    s' &= 2
\end{align*}
\]
Example of DFA Simulation

Let the input: "abababb"

begin
    s ← s₀;
    c ← nextChar;
    while c ≠ eof do
        s ← move (s, c);
        c ← nextChar;
    end
    return s ∈ F;
end

Example

s = 2
c = a
forward: babb
Example of DFA Simulation

Let the input: "abababb"

![DFA Diagram](image)

**Example**

```
begin
    s ← s₀;
    c ← nextChar;
    while c ≠ eof do
        s ← move (s, c);
        c ← nextChar;
    end
return s ∈ F;
end
```

- $s = 2$
- $c = a$
- $\text{move}(2, a) = 1$
- $s' = 1$
Example of DFA Simulation

Let the input: "abababb"

```
begin
  s ← s0;
c ← nextChar;
while c ≠ eof do
  s ← move (s, c);
c ← nextChar;
end
return s ∈ F;
end
```

**Example**

\[ s = 1 \]
\[ c = b \]
forward: \texttt{abb}
Example of DFA Simulation

Let the input: "abababb"

begin
  s ← s_0;
  c ← nextChar;
  while c ≠ eof do
    s ← move (s, c);
    c ← nextChar;
  end
  return s ∈ F;
end

Example

s = 1
\[\text{move}(1, b) = 2\]
\[s' = 2\]
Example of DFA Simulation

Let the input: ”abababb”

begin
    s ← s_0;
    c ← nextChar;
    while c ≠ eof do
        s ← move (s,c);
        c ← nextChar;
    end
    return s ∈ F;
end

Example

s = 2
\text{c = a}
\text{foward: bb}
Let the input: "abababb"

begin
s ← s_0;
c ← nextChar;
while c ≠ eof do
  s ← move (s, c);
  c ← nextChar;
end
return s ∈ F;
end

Example

s = 2
c = a
move(2, a) = 1
s' = 1
Example of DFA Simulation

Let the input: ”abababb”

begin
\[ s \leftarrow s_0; \]
\[ c \leftarrow \text{nextChar}; \]
\[ \textbf{while } c \neq \text{eof do} \]
\[ \quad s \leftarrow \text{move } (s, c); \]
\[ \quad c \leftarrow \text{nextChar}; \]
end
\[ \text{return } s \in F; \]

Example

\[ s = 1 \]
\[ c = b \]
\[ \text{foward: } b \]
Example of DFA Simulation

Let the input: ”abababb”

\[
\begin{align*}
\text{begin} & \quad s \leftarrow s_0; \\
& \quad c \leftarrow \text{nextChar}; \\
& \quad \textbf{while} \ c \neq \text{eof} \ \textbf{do} \\
& \qquad s \leftarrow \text{move} (s, c); \\
& \qquad c \leftarrow \text{nextChar}; \\
\textbf{end} & \quad \textbf{return} \ s \in F; \\
\text{end}
\end{align*}
\]

Example
\[
\begin{align*}
s &= 1 \\
c &= b \\
\text{move}(1, b) &= 2 \\
s' &= 2
\end{align*}
\]
Let the input: "abababb"

```
begin
    s ← s_0;
    c ← nextChar;
    while c ≠ eof do
        s ← move (s, c);
        c ← nextChar;
    end
return s ∈ F;
end
```

**Example**

\( s = 2 \)
\( c = b \)
\[\text{forward: } \text{eof}\]
Example of DFA Simulation

Let the input: ”abababb”

\begin{verbatim}
begin
  s ← s0;
  c ← nextChar;
  while c ≠ eof do
    s ← move (s, c);
    c ← nextChar;
  end
return s ∈ F;
end
\end{verbatim}

Example

\begin{align*}
  s &= 2 \\
  c &= b \\
  \text{move}(2, b) &= 3 \\
  s' &= 3
\end{align*}
Example of DFA Simulation

Let the input: "abababb"

begin
  s ← s0;
  c ← nextChar;
  while c ≠ eof do
    s ← move (s, c);
    c ← nextChar;
  end
return s ∈ F;
end

Example

s = 3
c = eof
Example of DFA Simulation

Let the input: ”abababb”

begin
    s ← s_0;
    c ← nextChar;
    while c ≠ eof do
        s ← move (s, c);
        c ← nextChar;
    end
    return s ∈ F;
end

Example

s = 3
F = \{3\}
Return “true”
Outline

1. Introduction
2. Input buffering
3. Specification and recognition of tokens
4. Writing a lexical analyzer with Lex, Flex, JFlex, JavaCC
5. Writing a lexical analyzer by hand
   - Finite automata
     - Nondeterministic finite automata
     - Deterministic finite automata
     - From regular expression to NFA
     - From NFA to DFA
   - Building a Lexical Analyzer
Algorithm of McNaughton-Yamada-Thompson (#1)

- **INPUT**: A regular expression \( r \) over alphabet \( S \).
- **OUTPUT**: An NFA \( N \) accepting \( L(r) \).
- **METHOD**: Begin by parsing \( r \) into its constituent subexpressions. The rules for constructing an NFA consist of basis rules for handling subexpressions with no operators, and inductive rules for constructing larger NFA from the NFAs for the immediate subexpressions of a given expression.

- **BASIS 1**: For each \( \epsilon \) in \( r \), construct the following NFA:
**BASIS 2:** For any subexpression $a$ in $\Sigma$, construct the following NFA:

![NFA Diagram]

Note that in both of the basis constructions, we construct a distinct NFA, with new states, for every occurrence of $\epsilon$ or some $a$ as a subexpression of $r$. 
Algorithm of McNaughton-Yamada-Thompson (#3)

**INDUCTION 1:** Suppose \( r = s | t \). Then \( N(r) \) is:

![Diagram for INDUCTION 1](image)

**INDUCTION 2:** Suppose \( r = st \). Then \( N(r) \) is:

![Diagram for INDUCTION 2](image)
**Algorithm of McNaughton-Yamada-Thompson (#4)**

- **INDUCTION 3**: Suppose $r = s^*$. Then $N(r)$ is:

![Diagram showing the algorithm of McNaughton-Yamada-Thompson](attachment:image.png)

- **INDUCTION 4**: Suppose $r = (s)$. Then $N(r) = N(s)$. 

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
Example of Conversion of $(a|b)^* abb$
Example of Conversion of $(a | b)^* abb$
Example of Conversion of \((a|b) \ast a bb\)
Example of Conversion of \((a|b)^* abb\)

```
\[ \begin{align*}
\text{start} & \rightarrow 7 \\
7 & \rightarrow 5 \\
5 & \rightarrow 1, 3, 6 \\
1 & \rightarrow 2 \\
2 & \rightarrow 4, 5 \\
4 & \rightarrow 3, 6 \\
3 & \rightarrow 5 \\
6 & \rightarrow 8 \\
8 & \rightarrow 9, 11, 5 \\
9 & \rightarrow 8, 10 \\
10 & \rightarrow 5 \\
\end{align*} \]
```

```
1 a \\
3 b \\
5 \\
4 \\
2 \\
7 6 \\
```

\[ r_1 \quad r_2 \quad \text{etc.} \]

\[ r_3 ( ) \]

\[ r_4 * \]

\[ r_5 \]

\[ r_6 \quad r_7 \quad r_8 \quad b \]

\[ r_9 \quad r_10 \]

\[ r_{11} \]

\[ a \]

\[ b \]

\[ ( \]

\[ ) \]
Introduction

2 Input buffering

3 Specification and recognition of tokens

4 Writing a lexical analyzer with Lex, Flex, JFlex, JavaCC

5 Writing a lexical analyzer by hand
   ■ Finite automata
      ■ Nondeterministic finite automata
      ■ Deterministic finite automata
      ■ From regular expression to NFA
   ■ From NFA to DFA
   ■ Building a Lexical Analyzer
The general idea behind the conversion is that each state of the constructed DFA corresponds to a set of NFA states.

It is possible that the number of DFA states is exponential in the number of NFA states, which could lead to difficulties when we try to implement this DFA.

The subset construction is an algorithm based on this idea. It is presented in the next slide.
Algorithm for Converting NFA to DFA

**Input**: An NFA $N$.

**Output**: A DFA $D$ accepting the same language $N$.

**Method**:

1. The algorithm constructs a transition table $D_{tran}$ from $D$. Each state of $D$ is a set of NFA states, and we construct $D_{tran}$ so that $D$ will simulate “in parallel” all the possible moves $N$ can make on a given input string.

2. The NDA may be built from the table $D_{tran}$. 
begin
  \( T \leftarrow \varepsilon\text{-closure}(s_0); \)
  \( D\text{states} \leftarrow \{T\}; \)
  \( \text{Unmarked} \leftarrow \{T\}; \)
  while \( \exists T \in \text{Unmarked} \) do
    \( \text{Unmarked} \leftarrow \text{Unmarked} \setminus \{T\}; \)
    foreach input symbol \( a \) do
      \( U \leftarrow \varepsilon\text{-closure}(\text{move}(T,a)); \)
      if \( U \notin D\text{States} \) then
        \( D\text{states} \leftarrow D\text{states} \cup \{U\}; \)
        \( \text{Unmarked} \leftarrow \text{Unmarked} \cup \{U\}; \)
      end
      \( D\text{tran}[T,a] \leftarrow U; \)
    end
  end
end
Let consider the NFA for the regular expression \((a|b)^* abb\).

Let consider the NFA for the regular expression \((a|b)^* abb\).
Let consider the NFA for the regular expression \((a|b)^* abb\).

![NFA Diagram]

<table>
<thead>
<tr>
<th>Label</th>
<th>Dstates</th>
<th>∈ Unmarked</th>
<th>&quot;a&quot;</th>
<th>&quot;b&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>{1, 3, 5, 7, 8}</td>
<td></td>
<td></td>
<td>B</td>
</tr>
<tr>
<td>B</td>
<td>{1, 2, 3, 5, 6, 8, 9}</td>
<td>×</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Notes

\(T = \{1, 3, 5, 7, 8\}\) and unmark \(T\)
\(a = "a"\)
\(U = \epsilon\text{-closure}(move(T, a)) = \epsilon\text{-closure}\{2, 9\} = \{1, 2, 3, 5, 6, 8, 9\}\)
\(U\) is a new state (B), and \(Dtran[T, a] = B\)
Let consider the NFA for the regular expression \((a|b)^* abb\).

**Example of the Building of** \(Dtran\)

\[\text{Label} \quad \text{\textit{Dstates}} \quad \in \quad \text{\textit{Unmarked}} \quad "a" \quad "b"
\]

| \(A\) | \{1, 3, 5, 7, 8\} | \in \{1, 3, 4, 5, 6, 8\} | \(B\) | \(C\) |
| \(B\) | \{1, 2, 3, 5, 6, 8, 9\} | \(\times\) |
| \(C\) | \{1, 3, 4, 5, 6, 8\} | \(\times\) |

**Notes**

\(T = \{1, 3, 5, 7, 8\}\)
\(a = "b"\)
\(U = \epsilon\)-closure(move\((T, a)\)) = \epsilon\)-closure\((\{4\})\) = \{1, 3, 4, 5, 6, 8\}
\(U\) is a new state (C), and \(Dtran[T, a] = C\)
Let consider the NFA for the regular expression \((a|b)^*abb\).

\[
\begin{align*}
tag{start} & \xrightarrow{} 7 \xrightarrow{} 5 \xrightarrow{\varepsilon} 1 \xrightarrow{a} 2 \xrightarrow{\varepsilon} 6 \xrightarrow{\varepsilon} 8 \xrightarrow{a} 9 \xrightarrow{b} 10 \xrightarrow{b} 11
\end{align*}
\]

<table>
<thead>
<tr>
<th>Label</th>
<th>(D\text{states})</th>
<th>(\varepsilon\text{ Unmarked})</th>
<th>&quot;a&quot;</th>
<th>&quot;b&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>{1, 3, 5, 7, 8}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>{1, 2, 3, 5, 6, 8, 9}</td>
<td></td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>C</td>
<td>{1, 3, 4, 5, 6, 8}</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Notes

\(T = \{1, 2, 3, 5, 6, 8, 9\}\) and unmark \(T\)
\(a = "a"\)
\(U = \varepsilon\text{-closure}(\text{move}(T, a)) = \varepsilon\text{-closure}(\{2, 9\}) = \{1, 2, 3, 5, 6, 8, 9\}\)
\(U\) is B, \(D\text{tran}[T, a] = B\)
Let consider the NFA for the regular expression \((a|b) \ast abb\).

![NFA Diagram]

<table>
<thead>
<tr>
<th>Label</th>
<th>Dstates</th>
<th>∈ Unmarked</th>
<th>&quot;a&quot;</th>
<th>&quot;b&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>{1, 3, 5, 7, 8}</td>
<td></td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>B</td>
<td>{1, 2, 3, 5, 6, 8, 9}</td>
<td></td>
<td>B</td>
<td>D</td>
</tr>
<tr>
<td>C</td>
<td>{1, 3, 4, 5, 6, 8}</td>
<td>×</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>{1, 3, 4, 5, 6, 8, 10}</td>
<td>×</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Notes**

\[ T = \{1, 2, 3, 5, 6, 8, 9\} \]

\[ a = "b" \]

\[ U = \varepsilon\text{-closure} (\text{move}(T, a)) = \varepsilon\text{-closure} (\{4, 10\}) = \{1, 3, 4, 5, 6, 8, 10\} \]

\[ U \text{ is a new state (D), } Dtran[T, a] = D \]
Example of the Building of $D_{tr}$

Let consider the NFA for the regular expression $(a|b)^* abb$.

![NFA Diagram]

<table>
<thead>
<tr>
<th>Label</th>
<th>$D_{states}$</th>
<th>$\in$ Unmarked</th>
<th>&quot;a&quot;</th>
<th>&quot;b&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>${1, 3, 5, 7, 8}$</td>
<td></td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>B</td>
<td>${1, 2, 3, 5, 6, 8, 9}$</td>
<td></td>
<td>B</td>
<td>D</td>
</tr>
<tr>
<td>C</td>
<td>${1, 3, 4, 5, 6, 8}$</td>
<td></td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>${1, 3, 4, 5, 6, 8, 10}$</td>
<td>$\times$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Notes**

$T = \{1, 3, 4, 5, 6, 8\}$ and unmark $T$

$a = "a"

$U = \epsilon$-closure(move($T$, a)) = $\epsilon$-closure($\{2, 9\}$) = $\{1, 2, 3, 5, 6, 8, 9\}$

$U$ is B, $D_{tr}[T, a] = B$
Example of the Building of \textit{Dtran}

Let consider the NFA for the regular expression \((a|b)^* abb\).

![NFA Diagram]

<table>
<thead>
<tr>
<th>Label</th>
<th>\textit{Dstates}</th>
<th>(\epsilon) Unmarked</th>
<th>&quot;a&quot;</th>
<th>&quot;b&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>{1, 3, 5, 7, 8}</td>
<td></td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>B</td>
<td>{1, 2, 3, 5, 6, 8, 9}</td>
<td></td>
<td>B</td>
<td>D</td>
</tr>
<tr>
<td>C</td>
<td>{1, 3, 4, 5, 6, 8}</td>
<td></td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>D</td>
<td>{1, 3, 4, 5, 6, 8, 10}</td>
<td>\times</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Notes**

\(T = \{1, 3, 4, 5, 6, 8\}\)
\(a = "b"\)

\(U = \epsilon\)-closure(move\((T, a)\)) = \epsilon\)-closure\((\{4\}) = \{1, 3, 4, 5, 6, 8\}\)

\(U\) is \(C\), \textit{Dtran}\([T, a]\) = \(C\)
Let consider the NFA for the regular expression \((a|b)^*abb\).

\[
\begin{array}{cccc}
\text{Label} & \text{Dstates} & \in \text{Unmarked} & \text{"a"} & \text{"b"} \\
A & \{1, 3, 5, 7, 8\} & & B & C \\
B & \{1, 2, 3, 5, 6, 8, 9\} & & B & D \\
C & \{1, 3, 4, 5, 6, 8\} & & B & C \\
D & \{1, 3, 4, 5, 6, 8, 10\} & \times & & B \\
\end{array}
\]

Notes

\(T = \{1, 3, 4, 5, 6, 8, 10\}\) and unmark \(T\)
\(a = "a"\)
\(U = \epsilon\)-closure(move(\(T, a\))) = \epsilon\)-closure(\(\{2, 9\}\)) = \(\{1, 2, 3, 5, 6, 8, 9\}\)
\(U\) is B, \(Dtran[T, a] = B\)
Let consider the NFA for the regular expression \((a|b)^* abb\).

Let's analyze the NFA step by step:

1. **Label**
   - **A**: \{1, 3, 5, 7, 8\}
   - **B**: \{1, 2, 3, 5, 6, 8, 9\}
   - **C**: \{1, 3, 4, 5, 6, 8\}
   - **D**: \{1, 3, 4, 5, 6, 8, 10\}
   - **E**: \{1, 3, 4, 5, 6, 8, 11\}

2. **Unmarked**
   - \(\epsilon\)-closure of \(T\) is \{1, 3, 4, 5, 6, 8, 10\}
   - \(a = "b"\)
   - \(U = \epsilon\)-closure of \(move(T, a)\) is \{4, 11\}
   - \(U\) is a new state, so \(Dtran[T, a] = E\)

**Notes**

- \(T = \{1, 3, 4, 5, 6, 8, 10\}\)
- \(a = "b"\)
- \(U = \epsilon\)-closure of \(move(T, a)\) is \{4, 11\}
- \(U\) is a new state, so \(Dtran[T, a] = E\)
Let consider the NFA for the regular expression \((a|b)^* abb\).

![Example of the Building of Dtran](image)

<table>
<thead>
<tr>
<th>Label</th>
<th>Dstates</th>
<th>(\in) Unmarked</th>
<th>&quot;a&quot;</th>
<th>&quot;b&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>{1, 3, 5, 7, 8}</td>
<td></td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>B</td>
<td>{1, 2, 3, 5, 6, 8, 9}</td>
<td></td>
<td>B</td>
<td>D</td>
</tr>
<tr>
<td>C</td>
<td>{1, 3, 4, 5, 6, 8}</td>
<td></td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>D</td>
<td>{1, 3, 4, 5, 6, 8, 10}</td>
<td></td>
<td>B</td>
<td>E</td>
</tr>
<tr>
<td>E</td>
<td>{1, 3, 4, 5, 6, 8, 11}</td>
<td></td>
<td>B</td>
<td></td>
</tr>
</tbody>
</table>

Notes:

\(T = \{1, 3, 4, 5, 6, 8, 11\}\) and unmark \(T\)
\(a = "a"\)

\(U = \epsilon\)-closure(move\((T, a)\)) = \epsilon\)-closure\((\{2, 9\}) = \{1, 2, 3, 5, 6, 8, 9\}\)

\(U\) is B, \(Dtran[T, a] = B\)
Let consider the NFA for the regular expression \((a|b)^* abb\).

\[
\begin{array}{c}
\text{start} \quad 7 \\
\text{1} \quad \varepsilon \\
\text{2} \quad a \\
\text{3} \quad b \\
\text{4} \quad \varepsilon \\
\text{5} \quad \varepsilon \\
\text{6} \quad \varepsilon \\
\text{8} \quad \varepsilon \\
\text{9} \quad a \\
\text{10} \quad b \\
\text{11} \quad \varepsilon \\
\end{array}
\]

<table>
<thead>
<tr>
<th>Label</th>
<th>Dstates</th>
<th>∈ Unmarked</th>
<th>&quot;a&quot;</th>
<th>&quot;b&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>{1, 3, 5, 7, 8}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>{1, 2, 3, 5, 6, 8, 9}</td>
<td></td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>C</td>
<td>{1, 3, 4, 5, 6, 8}</td>
<td></td>
<td>B</td>
<td>D</td>
</tr>
<tr>
<td>D</td>
<td>{1, 3, 4, 5, 6, 8, 10}</td>
<td></td>
<td>B</td>
<td>E</td>
</tr>
<tr>
<td>E</td>
<td>{1, 3, 4, 5, 6, 8, 11}</td>
<td></td>
<td>B</td>
<td>C</td>
</tr>
</tbody>
</table>

Notes:

\(T = \{1, 3, 4, 5, 6, 8, 11\}\)
\(a = "b"\)
\(U = \varepsilon\text{-closure}(\text{move}(T, a)) = \varepsilon\text{-closure}(\{4\}) = \{1, 3, 4, 5, 6, 8\}\)
\(U\) is C, \(Dtran[T, a] = C\)
**Building the DFA from the Table** \(D_{\text{tran}}\)

<table>
<thead>
<tr>
<th>Label</th>
<th>(D_{\text{states}})</th>
<th>Init.</th>
<th>Final</th>
<th>&quot;,a&quot;</th>
<th>&quot;,b&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>{1, 3, 5, 7, 8}</td>
<td>yes</td>
<td>(\emptyset)</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>B</td>
<td>{1, 2, 3, 5, 6, 8, 9}</td>
<td>no</td>
<td>(\emptyset)</td>
<td>B</td>
<td>D</td>
</tr>
<tr>
<td>C</td>
<td>{1, 3, 4, 5, 6, 8}</td>
<td>no</td>
<td>(\emptyset)</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>D</td>
<td>{1, 3, 4, 5, 6, 8, 10}</td>
<td>no</td>
<td>(\emptyset)</td>
<td>B</td>
<td>E</td>
</tr>
<tr>
<td>E</td>
<td>{1, 3, 4, 5, 6, 8, 11}</td>
<td>no</td>
<td>{11}</td>
<td>B</td>
<td>C</td>
</tr>
</tbody>
</table>

![Diagram of the DFA](image-url)
Outline

1. Introduction
2. Input buffering
3. Specification and recognition of tokens
4. Writing a lexical analyzer with Lex, Flex, JFlex, JavaCC
5. Writing a lexical analyzer by hand
   - Finite automata
     - Building a Lexical Analyzer
       - Pattern matching with NFA
       - Pattern matching with DFA
To construct the automaton, we take each regular-expression pattern and converting it to an NFA.

We need a single global automaton, so we combine all the NFA’s into one by introducing a new start state.

Let take the example:

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>do_Action1();</td>
</tr>
<tr>
<td>abb</td>
<td>do_Action2();</td>
</tr>
<tr>
<td>a*b+</td>
<td>do_Action3();</td>
</tr>
</tbody>
</table>
Example of the NFA Automaton

\[
\begin{align*}
\text{a} & \quad \text{start} & 1 & a & 2 \\
\text{abb} & \quad \text{start} & 3 & a & 4 & b & 5 & b & 6 \\
\text{a*b+} & \quad \text{start} & 7 & a & 8 & b
\end{align*}
\]
Example of the NFA Automaton

\[ \begin{array}{cccccc}
\text{start} & \varepsilon & 0 & \varepsilon & 3 & \varepsilon \\
0 & \varepsilon & 1 & a & 2 & a \\
3 & a & 4 & b & 5 & b \\
7 & b & 6 & b & 6 & b \\
\end{array} \]
Running the NFA Automaton

- The lexical analyzer reads the input from lexemeBegin.
- The NFA is evaluated according to the input pointed by the forward pointer.

- When the NFA simulation does not find any more state, we could find the longest validated lexeme:
  - Look backwards in the sequence of sets of states, until accepting states were found.
  - If found accepting states, replies the associated lexeme.
  - Otherwise, there is a syntax error.
Example of Simulation of NFA

Let consider the input: aaba

Initially, the set of states contains the \( \epsilon \)-closure of the state 0.
Let consider the input: aaba

Read "a"
States: \( \varepsilon\text{-closure}(\text{move}([0, 1, 3, 7], "a")) = \{2, 4, 7\} \)
State 2 is a final state \( \Rightarrow \) lexeme detected for pattern a
Let consider the input: aaba

Read "a"
States: $\epsilon$-closure(move({2, 4, 7}, "a")) = {7}
Example of Simulation of NFA

Let consider the input: aaba

Read "b"
States: $\epsilon$-closure(move($\{7\}$, "b")) = $\{8\}$
State 8 is a final state $\Rightarrow$ lexeme detected for pattern $a^*b+$
Example of Simulation of NFA

Let consider the input: aaba

Read "a"
States: $\epsilon$-closure(move({8}, "a")) = $\emptyset$
Simulation is done. Look backward.
Let consider the input: aaba

Longuest lexeme: "aab"
Matching pattern: a*b+
Execute do_Action3()
Outline

1. Introduction
2. Input buffering
3. Specification and recognition of tokens
4. Writing a lexical analyzer with Lex, Flex, JFlex, JavaCC
5. Writing a lexical analyzer by hand
   - Finite automata
   - Building a Lexical Analyzer
     - Pattern matching with NFA
     - Pattern matching with DFA
To construct the automaton, we take each regular-expression pattern and converting it to an DFA (directly or via a NFA).

Within each DFA state, if there are one or more accepting NFA states, use the first pattern in the Lex program associated to these NFA states.

Let take the example:

```
a { do_Action1(); }
abb { do_Action2(); }
a*b+ { do_Action3(); }
```
Example of the DFA Automaton

Note

Both states 6 and 8 are final states for patterns “abb” and “a*b+”, resp. Only the first in the Lex program is considered by the DNA.
Running the DFA Automaton

- We use the DFA in a lexical analyzer much as we did with the NFA.

- We simulate the DFA until at some point there is no next state, or strictly speaking, the next state is $\emptyset$, the dead state corresponding to the empty set of NFA states.

- At that point, we go back through the sequence of states.

- As soon as an accepting DFA state is encountered, the associated action is performed.
Example of Simulation of DFA

Let consider the input: aaba

Initially, the selected state is (0137).
Let consider the input: aaba

Example of Simulation of DFA

Read "a".
Pass the edge of the DFA, and update the current state.
Because the state 2 is a final state in the NFA, the state (247) is marked.
Let consider the input: aaba

Example of Simulation of DFA

Read "a", and pass the edge.
Example of Simulation of DFA

Let consider the input: aaba

Read "b".
Pass the edge of the DFA, and update the current state.
Because the state 8 is a final state in the NFA, the state (8) is marked.
Let consider the input: aaba

Example of Simulation of DFA

Read "a".
No state is accessible.
Simulation is done. Look backward.
Let consider the input: aaba

Longuest lexeme: "aab"
Matching pattern: a*b+
Execute do_Action3()
Outline

1. Introduction
2. Input buffering
3. Specification and recognition of tokens
4. Writing a lexical analyzer with Lex, Flex, JFlex, JavaCC
5. Writing a lexical analyzer by hand
6. Conclusion
Tokens: The lexical analyzer scans the source program and produces as output a sequence of tokens, which are normally passed, one at a time to the parser.

Lexemes: Each time the lexical analyzer returns a token to the parser, it has an associated lexeme: the sequence of characters that the token represents.

Buffering: Because it is often necessary to scan ahead on the input in order to see where the next lexeme ends, it is necessary for the lexical analyzer to buffer the input.

Patterns: Each token has a pattern that describes which sequences of characters can form the lexemes corresponding to that token.

Regular Expressions: These expressions are commonly used to describe patterns. Regular expressions are built from single characters, using union, concatenation, and the Kleene closure.
Transition Diagram: The behavior of a lexical analyzer can be described with a transition diagram. The states of that diagram represent the history of the characters seen during the analysis. The edges between the states indicate the possible next characters.

Finite Automata: These are a formalization of transition diagrams. Accepting states indicates that a lexeme for a token has been found. Unlike transition diagrams, finite automata can make transitions on empty input as well as on input characters.

Deterministic Finite Automata: A DFA is a special kind of finite automata that has exactly one transition out from each state for each input symbol.

Nondeterministic Finite Automata: Automata that are not DFA are called nondeterministic.
Conversion Among Pattern Representations: It is possible to convert any regular expression to NFA, and to convert any NFA to DFA.

Lex: Family of software systems that are able to generate lexical analyzers from input specifications.


Chapter 3
Syntax Analysis

Stéphane GALLAND
1 Introduction

2 Context-free grammar

3 Parsing with a grammar

4 Generate a syntactic parser with Yacc or JavaCC

5 Conclusion
1 Introduction
   - General principles
   - Error recovery

2 Context-free grammar

3 Parsing with a grammar

4 Generate a syntactic parser with Yacc or JavaCC

5 Conclusion
This chapter is devoted to the parsing methods that are typically used in compilers.

By design, every programming language has precise rules that prescribe the syntactic structure of well-formed programs.

The syntax of programming language constructs can be specified by context-free grammars or BNF (Backus-Naur Form.)

The output of the syntax analyzer is a parse tree for the stream of tokens that comes from the lexical analyzer.
A grammar gives a precise, easy-to-understand, syntactic specification of a programming language.

From certain classes of grammars, we can construct automatically an efficient parser that determine the syntactic structure of a source program.

The structure imparted to a language by a properly designed grammar is useful for translating source programs into correct object code and for detecting errors.

A grammar allows a language to be evolved or developed iteratively, by adding new constructs to perform new tasks.
The syntax analyzer, through the parser, generally controls a part of the execution flow of the compiler.

There are three types of parsers for grammars: universal, top-down, bottom-up.

But universal methods, such as the Cocke-Younger-Kasami algorithm and Earley’s algorithm, are too inefficient.
The most efficient top-down and bottom-up methods work only for subclasses of grammars: $LL$ and $LR$ grammars.  

But $LL$ and $LR$ grammars are expressive enough to describe most of the syntactic constructs in modern programming languages.

$LL$ grammars are commonly used when writing a parser by hands  

$LR$ grammars are more complex and used in parser generators.
A compiler is expected to assist the programmer in locating and tracking down errors that inevitably creep into programs.

Most programming language specifications do not describe how a compiler should respond to errors; error handling is left to the compiler designer.

Why handling errors during the parsing?:

- LL and LR methods permits to detect errors efficiently and as soon as possible.
- Many errors appear syntactic, whatever they cause, and are exposed when parsing cannot continue.

The error handler must:

- Report the presence of errors clearly and accurately.
- Recover from each error quickly to detect subsequent errors.
- Add minimal overhead to the processing of correct programs.
Common Programming Errors

- **Lexical errors**: they include misspellings of identifiers, keywords, or operators; and missing quotes around text intended as a string.

- **Syntactic errors**: they include misplaced semicolons or extra or missing braces. Another example is a `case` outside an enclosing `switch` block.

- **Semantic errors**: they include type mismatches between operators and operands.

- **Logical errors**: they can be anything from incorrect reasoning on the part of the programmer to the use in a program. For example, the use of the operator `"=="` in place of the operator `"=="`; or unreachable code.
Once an error is detected, how should the parser recover?

Although no strategy has proven itself universally acceptable.

The simplest approach is for a parser to quit with an informative error message when it detects the first error; additional errors are uncovered.

If errors pile up, it is better for the compiler to give up after exceeding some error limit than to produce an annoying avalanche of “spurious” errors.

The rest of this section is devoted to the two major error-recovery strategies: panic mode, and phrase-level recovery.
The parser discards input symbols one at a time until one of a designated set of synchronizing tokens is found.

The synchronizing tokens are usually delimiters, such as semicolons or closing braces, whose role in the source program is clear and unambiguous.

This approach is simple to implement and it guarantees not to go into an infinite loop.
On discovering an error, the parser may perform local correction on the remaining input:

- It may replace the prefix of the remaining input by some string that it allows the parser to continue; eg. replacing a comma by a semicolon, remove extraneous semicolon; or insert a missed semicolon.

- The choice of the correction is left to the compiler designer.

- The major drawback of the phrase-level recovery is the difficulty it has in coping with situations in which the actual error has occurred before the point of detection.
To anticipate the error detection, we can augment the grammar for the language at hand with productions that generate the erroneous constructs.

A parser constructed from an augmented grammar detects the anticipated errors when an error production is used during parsing.

The parser can then generate appropriate error diagnostics about the erroneous constructs that has been recognized in the input.
It would be helpful that a compiler makes few changes as possible in processing an incorrect input string.

Given an incorrect input string $x$ and grammar $G$, some algorithms find a parse tree for a related string $y$, such that the number of insertions, deletions, and changes of tokens required to transform $x$ to $y$ is as small as possible.

Unfortunately, these methods are too costly in time and space.

Global corrections has been used to evaluate error-recovery algorithms and to find optimal replacement strings for phrase-level recovery.
Outline

1. Introduction

2. Context-free grammar
   - Definition and notation
   - Derivations and Parse Tree
   - Ambiguity of a grammar
   - Verifying the language supported by a grammar
   - Context-free grammar and regular expression
   - Guidelines for writing a grammar

3. Parsing with a grammar

4. Generate a syntactic parser with Yacc or JavaCC
Grammars are systematically used to describe the syntax of programming language constructs.

- A context-free grammar consists of terminals, nonterminals, a start symbol, and productions.

**Definition (Terminal — Token Name)**

The basic symbols from which strings are formed. It could be assimilated to a token, replied by the lexical analyzer (see chapter 81).
Definition (Nonterminals)

Syntactic variables that denote sets of strings. The sets of strings denoted by nonterminals help to define the language generated by the grammar. Nonterminals impose a hierarchical structure on the language that is key to syntax analysis and translation.
Definition (Production)

The productions of a grammar specify the manner in which the terminals and nonterminals can be combined to form strings. Each production consists of:

1. A nonterminal called the head or left side of the production; this production defines some of the strings denoted by the head.
2. The symbol "→" (or "::=").
3. A body, or right side, consisting of zero or more terminals and nonterminals. The components of the body describe one way in which strings of the nonterminal at the head can be constructed.
Definition (Start Symbol)

In a grammar, one nonterminal is distinguished as the start symbol, and the set of strings it denotes is the language generated by the grammar. Conventionally, the productions for the start symbol are listed first.
General Principles

1. A grammar derives strings by beginning with the start symbol.
2. It is repeatedly replacing a nonterminal by the body of a production for that nonterminal.
3. The terminal strings that can be derived form the language defined by the grammar.

Parsing is the problem of taking a string of terminals and figuring out how to derive it from the start symbol of the grammar. If the string cannot be derived, the parser reports a syntax error.
These symbols are terminals:

- Lowercase letters early in the alphabet such as a, b, c, ...
- Operator symbols, such as +, *, ...
- Punctuation symbols, such as parentheses, commas, ...
- The digits 0, ..., 9.
- Boldface strings, such as \textit{id} or \texttt{number}.
- Underlined strings, such as \underline{id} or \underline{number}.

These symbols are nonterminals:

- Uppercase letters, such as A, B, C, ...
- The letter $S$ which, when it appears, is usually the start symbol.
- Lowercase, italic names such as \textit{expression}, \textit{factor}, ...
Conventions on Notation (§2)

- **Productions:** A set of productions
  \[ A \rightarrow a_1, A \rightarrow a_2, \ldots, A \rightarrow a_k \]
  with a common head \( A \) (call them \( A \)-productions), may be written
  \[ A \rightarrow a_1 | a_2 | \ldots | a_k \]
  the alternatives of \( A \).

- **Start Symbol:** Unless stated otherwise, the head of the first production is the start symbol.

- **Others Notations:**
  - Uppercase letters late in the alphabet, such as \( X, Y, Z \), represent grammar symbols that is, either nonterminals or terminals.
  - Lowercase letters late in the alphabet, chiefly \( u, v, \ldots, z \), represent (possibly empty) strings of terminals.
  - Lowercase Greek letters \( \alpha, \beta, \ldots \), represent (possibly empty) strings of grammar symbols.
The arithmetic expressions are defined by the following grammar.

The terminals are:

- **Operators**: +, -, *, /, (, );
- **Numbers**: `number` stands for any number;
- **Identifier**: `id` stands for any variable’s name.

\[
\begin{align*}
\text{expression} & \rightarrow \text{expression} + \text{term} \\
& \quad | \quad \text{expression} - \text{term} \\
\text{term} & \rightarrow \text{term} * \text{factor} \\
& \quad | \quad \text{term} / \text{factor} \\
& \quad | \quad \text{factor} \\
\text{factor} & \rightarrow ( \text{expression} ) \\
& \quad | \quad \text{number} \\
& \quad | \quad \text{id}
\end{align*}
\]
1 Introduction

2 Context-free grammar
   - Definition and notation
   - Derivations and Parse Tree
     - Derivations
     - Parse Tree
     - Building a Parse Tree with Derivations
   - Ambiguity of a grammar
   - Verifying the language supported by a grammar
   - Context-free grammar and regular expression
   - Guidelines for writing a grammar

3 Parsing with a grammar
Derivations

- From the start symbol, each rewriting step replaces a nonterminal by the body of one of its productions.
- Let the example:

  \[ E \rightarrow E + E \mid E * E \mid - E \mid (E) \mid \text{id} \]

- The replacement of \( E \) by \(- E\) will be described by writing:

  \[ E \Rightarrow - E \]

- The symbol \( \Rightarrow \) means “derives in one step”.

Example

- Let a nonterminal \( A \) in the middle of symbols: \( \alpha A \beta \)
- Let the production: \( A \rightarrow \gamma \)
- Then: \( \alpha A \beta \Rightarrow \alpha \gamma \beta \)
Definition

- The sequence of derivations \( a_1 \Rightarrow a_2 \Rightarrow \ldots \Rightarrow a_n \) rewrites \( a_1 \) to \( a_n \).
- It may also be written: \( a_1 \Rightarrow^* a_n \).

Properties

1. \( \alpha \Rightarrow^* \alpha \), for any string \( \alpha \), and
2. If \( \alpha \Rightarrow^* \beta \), and \( \beta \Rightarrow \gamma \), then \( \alpha \Rightarrow^* \gamma \).
If $S \rightarrow a$, where $S$ is the start symbol of a grammar $G$, we say that $a$ is a sentential form of $G$.

A sentence of $G$ is a sentential form, which is nonterminal.

The language generated by $G$ is its set of sentences.

A string of terminals $w$ is in $L(G)$ iff $S \rightarrow^* w$. Thus $L(G)$ is said to be a context-free language.
At each step in a derivation, there are two choices to be made:

1. To choose which nonterminal to replace, and
2. To pick a production with that nonterminal as head.

To understand how parsers work, we shall consider derivations in which the nonterminal to be replaced at each step is chosen as follows:

- **Leftmost derivation**: the leftmost nonterminal in each sentential is always chosen. If \( \alpha \Rightarrow \beta \) is a step in which the leftmost nonterminal in \( \alpha \) is replaced, we write \( \alpha \Rightarrow_{lm} \beta \).

- **Rightmost derivation**: the rightmost nonterminal is always chosen, we write \( \alpha \Rightarrow_{rm} \beta \).
Example of Leftmost Derivations

Let the grammar:

\[
E \rightarrow E + E \\
| \quad E \ast E \\
| \quad - E \\
| \quad ( E ) \\
| \quad id
\]

Let the input string: 2 + 4 * 6

Corresponding list of tokens: id+id*id

The leftmost derivations are:

\[
E \Rightarrow E + E \\
\downarrow \quad lm \\
E + E \Rightarrow E + E
\]
Example of Leftmost Derivations

Let the grammar:

\[
E \rightarrow E + E \\
| E * E \\
| - E \\
| ( E ) \\
| id
\]

Let the input string: 2 + 4 * 6

Corresponding list of tokens: id+id*id

The leftmost derivations are:

\[
E \Rightarrow E + E \\
\Rightarrow id + E
\]
Example of Leftmost Derivations

Let the grammar:

\[
E \rightarrow E + E \\
| \quad E \ast E \\
| \quad - E \\
| \quad (E) \\
| \quad id
\]

Let the input string: 2 + 4 * 6

Corresponding list of tokens: id + id * id

The leftmost derivations are:

\[
E \Rightarrow E + E \\
\Rightarrow id \ast E \\
\Rightarrow id + E \ast E
\]
Example of Leftmost Derivations

Let the grammar:

$$E \rightarrow E + E \quad | \quad E \ast E \quad | \quad - E \quad | \quad (E) \quad | \quad id$$

Let the input string: $2 + 4 \ast 6$

Corresponding list of tokens: id+id*id

The leftmost derivations are:

$$E \Rightarrow E + E \quad \Rightarrow \quad id + E \ast E \quad \Rightarrow \quad id + id \ast E$$
Example of Leftmost Derivations

Let the grammar:

\[
E \rightarrow E + E \\
| E \ast E \\
| - E \\
| ( E ) \\
| id
\]

Let the input string: 2 + 4 \ast 6

Corresponding list of tokens: id+id*id

The leftmost derivations are:

\[
\begin{align*}
E & \Rightarrow E + E \\
& \Rightarrow id + E \\
& \Rightarrow id + E \ast E \\
& \Rightarrow id + id \ast E \\
& \Rightarrow id + id \ast id \\
\end{align*}
\]
Example of Rightmost Derivations

Let the grammar:

<table>
<thead>
<tr>
<th>$E$</th>
<th>$→$</th>
<th>$E + E$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$</td>
<td>$</td>
</tr>
<tr>
<td></td>
<td>$</td>
<td>$</td>
</tr>
<tr>
<td></td>
<td>$</td>
<td>$</td>
</tr>
<tr>
<td></td>
<td>$</td>
<td>$</td>
</tr>
</tbody>
</table>

Let the input string: $2 + 4 * 6$

Corresponding list of tokens: $id + id * id$

The leftmost derivations are: $E \Rightarrow E + E$
Example of Rightmost Derivations

Let the grammar:

\[
E \rightarrow E + E \\
| E \ast E \\
| - E \\
| ( E ) \\
| id
\]

Let the input string: 2 + 4 * 6

Corresponding list of tokens: \text{id} + \text{id} * \text{id}

The leftmost derivations are:

\[
E \Rightarrow E + E \\
\Rightarrow E + E \ast E
\]
Example of Rightmost Derivations

- Let the grammar:

\[
E \rightarrow E + E \\
| E \ast E \\
| - E \\
| (E) \\
| id
\]

- Let the input string: 2 + 4 * 6

- Corresponding list of tokens: id + id * id

- The leftmost derivations are:

\[
E \Rightarrow E + E \\
\Rightarrow E + E \ast E \\
\Rightarrow E + E \ast id
\]
Example of Rightmost Derivations

Let the grammar:

\[ E \rightarrow E + E \]
\[ \quad | \quad E * E \]
\[ \quad | \quad - E \]
\[ \quad | \quad ( E ) \]
\[ \quad | \quad id \]

Let the input string: 2 + 4 * 6

Corresponding list of tokens: id + id * id

The leftmost derivations are:

\[ E \Rightarrow E + E \]
\[ \quad \Rightarrow E + E * E \]
\[ \quad \Rightarrow E + E * id \]
\[ \quad \Rightarrow E + id * id \]
Example of Rightmost Derivations

Let the grammar:

\[
E \rightarrow E + E \\
    | E \times E \\
    | - E \\
    | ( E ) \\
    | id
\]

Let the input string: 2 + 4 * 6

Corresponding list of tokens: id+id*id

The leftmost derivations are:

\[
E \Rightarrow E + E \\
    \Rightarrow E + E \times E \\
    \Rightarrow E + E \times id \\
    \Rightarrow E + id \times id \\
    \Rightarrow id + id \times id
\]
1 Introduction

2 Context-free grammar
   - Definition and notation
   - Derivations and Parse Tree
     - Derivations
     - Parse Tree
     - Building a Parse Tree with Derivations
   - Ambiguity of a grammar
   - Verifying the language supported by a grammar
   - Context-free grammar and regular expression
   - Guidelines for writing a grammar

3 Parsing with a grammar

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
Definition (Parse Tree)

A parse tree shows how the start symbol of a grammar derives a string. It is an pictorial representation of the productions on an input string of tokens.

Building a Parse Tree

- The root is labeled by the start symbol.
- Each leaf is labeled by a terminal or by $\epsilon$.
- Each interior node is labeled by a nonterminal.
- If $A$ is the nonterminal of some interior node and $X_1, X_2, \ldots, X_n$ are the labels of the children of that node from left to right, then there must be a production $A \rightarrow X_1X_2 \ldots X_n$.

Parsing is the process of building a parse tree.
Let the string to parse:

9 - 5 + 2

Let the grammar:

| expression  | → | expression + term |
|            |   | expression - term |
| term       | → | term * factor     |
|            |   | term / factor     |
| factor     | → | ( expression )    |
|            |   | number            |
|            |   | id                |

The parse tree is:

```
expression
  +
  term
    -
    term
      *
      factor
        /
        factor
          /
          factor
            /
            number:2
              /
              number:9
                /
                number:5
```
1 Introduction

2 Context-free grammar
   - Definition and notation
   - Derivations and Parse Tree
     - Derivations
     - Parse Tree
     - Building a Parse Tree with Derivations
   - Ambiguity of a grammar
   - Verifying the language supported by a grammar
   - Context-free grammar and regular expression
   - Guidelines for writing a grammar

3 Parsing with a grammar
Algorithm to Build a Parse Tree with Derivations

Input: A sequence of tokens $T$. A grammar $G$ with the start symbol $s_0$.
Output: A parse tree that corresponds to $T$ and $G$.

begin
  $r \leftarrow$ node ($s_0$, $T$); $L \leftarrow [r]$; $input[r] \leftarrow T$
  while $L = [n].L'$ do /* Leftmost derivation */
    $L \leftarrow L'$;
    if $\exists$ (label($n$) $\rightarrow$ $b$) $\in G$|$input[n]$ matches $b$ then
      foreach $\alpha|s|\beta = b$ do
        $m \leftarrow \omega \in T$|($input[\alpha] \omega input[\beta]$) $= input[n]$;
        $c \leftarrow$ node ($s$);
        addChild ($n$, $c$);
        if $s$ is nonterminal then
          $L \leftarrow L.[c]$;
          $input[c] \leftarrow m$;
        end
      end
    end
  end
return $r$;
end
Example of Parse Tree Building

Let the grammar:

\[
E \rightarrow \begin{align*}
E &+ E \\
E \times E \\
- E \\
( E ) \\
id
\end{align*}
\]

Tokens: \(id + id * id\)

\(L = [E]\)
Example of Parse Tree Building

Let the grammar:

\[
E \rightarrow E + E \\
| \quad E * E \\
| \quad - E \\
| \quad ( E ) \\
| \quad \text{id}
\]

Parse tree is:

\[
\begin{array}{c}
E \\
+ \\
E
\end{array}
\]

Tokens: \text{id}+\text{id}*\text{id}

\[
L = [n].L' = [E] \\
input = \text{id}+\text{id}*\text{id} \\
b = E_0 + E_1 \\
input_{E_0} = \text{id} \\
input_{E_1} = \text{id}*\text{id} \\
L = [E_0, E_1]
\]
Example of Parse Tree Building

Let the grammar:

\[
E \rightarrow E + E \\
| E \ast E \\
| - E \\
| ( E ) \\
| id
\]

Parse tree is:

\[
E \\
+ \\
E
\]

Tokens: \textit{id}+\textit{id}\ast\textit{id}

\[
L = [n]. L' = [E, E] \\
input = \textit{id} \\
b = \textit{id} \\
L = [E]
\]
Let the grammar:

\[
E \rightarrow E + E \\
\quad \quad E * E \\
\quad \quad - E \\
\quad \quad (E) \\
\quad \quad id
\]

Parse tree is:

\[
\begin{array}{c}
E \\
\quad + \\
\quad \quad E \\
\quad \quad \quad * \\
\quad \quad \quad \quad E \\
\quad \quad \quad \quad id
\end{array}
\]

Tokens: id+id*id

\[
L = [n].L' = [E] \\
input = id*id \\
b = E_0 * E_1 \\
input_{E_0} = id \\
input_{E_1} = id \\
L = [E_0, E_1]
\]
Example of Parse Tree Building

Let the grammar:

\[
E \rightarrow E + E \\
\mid E \ast E \\
\mid - E \\
\mid (E) \\
\mid \text{id}
\]

Parse tree is:

```
  E
   +
    E
     *
      E
       
       id
```

Tokens: \text{id} + \text{id} \ast \text{id}

\[
L = [n].L' = [E, E] \\
\text{input} = \text{id} \\
b = \text{id} \\
L = [E]
\]
Let the grammar:

\[
E \rightarrow E + E \\
| E \ast E \\
| - E \\
| ( E ) \\
| id
\]

Tokens: id + id * id

\[
L = [n].L' = [E] \\
input = id \\
b = id \\
L = []
\]
1 Introduction

2 Context-free grammar
   ▪ Definition and notation
   ▪ Derivations and Parse Tree
   ▪ Ambiguity of a grammar
   ▪ Verifying the language supported by a grammar
   ▪ Context-free grammar and regular expression
   ▪ Guidelines for writing a grammar

3 Parsing with a grammar

4 Generate a syntactic parser with Yacc or JavaCC
What is an Ambiguous Grammar?

A grammar that produces more than one parse tree for some sentence is said to be ambiguous.

- An ambiguous grammar is one that produces more than one leftmost derivation or more than one rightmost derivation for the same sentence.

Example

Leftmost derivations for the arithmetic expression \texttt{id+id*id}.
For parsers, it is desirable that the grammar be made unambiguous. Otherwise we cannot determine which parse tree to select for a sentence.

Another way is to use carefully chosen ambiguous grammars, together with disambiguating rules that discard undesirable parse trees, leaving only one tree for each sentence.
1 Introduction

2 Context-free grammar
   - Definition and notation
   - Derivations and Parse Tree
   - Ambiguity of a grammar
   - Verifying the language supported by a grammar
   - Context-free grammar and regular expression
   - Guidelines for writing a grammar

3 Parsing with a grammar

4 Generate a syntactic parser with Yacc or JavaCC
Verifying the Language Supported by a Grammar

- Even if compiler designers rarely do this task, it is useful to be able to verify if a language can be generated from a grammar.

- A proof that a grammar $G$ generates a language $L$ has two parts:
  1. Show that every string generated by $G$ is in $L$.
  2. Show that every string in $L$ can be generated by $G$.

**Example**

Considerer the following grammar:

$$S \rightarrow (\ S\ )\ S \mid e$$

It may not be apparent, but this grammar generates all the strings of balanced parentheses, and only such strings. That why, we need to proceed the two steps of the proof.
Part 1: every string generated by $G$ is in $L$

- **BASIS:** The basis is $n = 1$. The only string of terminals derivable from $S$ in one step is the empty string, which is balanced.

- **INDUCTION:** Assume that all derivations of fewer than $n$ steps produce balanced sentences, and consider a leftmost derivation of exactly $n$ steps. Such a derivation must be of the form:

  $$S \Rightarrow_{lm} ( S ) S \Rightarrow_{lm} ( x ) S \Rightarrow_{lm} ( x ) y$$

  The derivations of $x$ and $y$ from $S$ take fewer than $n$ steps, so by the inductive hypothesis $x$ and $y$ are balanced. Therefore, the string $(x)y$ must be balanced.
Part 2: every string in $L$ can be generated by $G$

- **BASIS**: If the string is length 0, it must be $\epsilon$, which is balanced.
- **INDUCTION**: Observe that every balanced string has even length. Assume that every balanced string of length less than $2n$ is derivable from $S$, and consider a balanced string $w$ of length $2n$, $n \geq 1$. Surely $w$ begins with a left parenthesis. Let $(x)$ be the shortest nonempty prefix of $w$ having an equal number of left and right parentheses. Then $w$ can be written $w = (x)y$ where both $x$ and $y$ are balanced. Since $x$ and $y$ are of length less than $2n$, they are derivable from $S$ by the inductive hypothesis. Thus, we can find a derivation of the form:
  
  $$S \Rightarrow (S)S^* \Rightarrow (x)S^* \Rightarrow (x)y$$

proving that $w = (x)y$ is also derivable from $S$. 
1 Introduction

2 Context-free grammar
   - Definition and notation
   - Derivations and Parse Tree
   - Ambiguity of a grammar
   - Verifying the language supported by a grammar
   - Context-free grammar and regular expression
   - Guidelines for writing a grammar

3 Parsing with a grammar

4 Generate a syntactic parser with Yacc or JavaCC
Grammars are more powerful notation than regular expressions.

Every construct that can be described by a regular expression can be described by a grammar, but not vice-versa.

For example the regular expression \((a|b)^*abb\) and the following grammar describe the same language.

\[
\begin{align*}
A_0 & \rightarrow a\ A_0 \mid b\ A_0 \mid a\ A_1 \\
A_1 & \rightarrow b\ A_2 \\
A_2 & \rightarrow b\ A_3 \\
A_3 & \rightarrow \epsilon
\end{align*}
\]

In the other hand, the language \(L = a^n b^n | n \geq 1\) is an example of a language that can be described by a grammar but not by a regular expression (except for Posix extension).
It is possible to construct a grammar from a regular expression through the corresponding NFA.

This mechanical transformation follows the steps below:

1. For each state $i$ of the NFA, create a nonterminal $A_i$.
2. If state $i$ has a transition to state $j$ on input $a$, add the production $A_i \rightarrow aA_j$. If state $i$ goes to state $j$ on input $\epsilon$, add the production $A_i \rightarrow A_j$.
3. If $i$ is an accepting state, add $A_i \rightarrow \epsilon$.
4. If $i$ is the state $s$, make $A_i$ be the start symbol of the grammar.
Outline

1. Introduction

2. Context-free grammar
   - Definition and notation
   - Derivations and Parse Tree
   - Ambiguity of a grammar
   - Verifying the language supported by a grammar
   - Context-free grammar and regular expression
   - Guidelines for writing a grammar
     - General guidelines
     - Eliminating the ambiguity
     - Eliminating the left recursion
     - Left factoring

3. Parsing with a grammar

4. Generate a syntactic parser with Yacc or JavaCC
Grammars are capable of describing most, but not all, of the syntax of programming languages.

The requirement that identifiers be declared before they are used cannot be described in a grammar.

As seen previously, everything that can be described with a regular expression, can also be described by a grammar.

Why use regular expressions to define the lexical syntax of a language?

1. Separating the syntactic structure of a language into lexical and non-lexical parts provides a better modularity.
2. The lexical rules of a language are frequently quite simple, and to describe them we do not need a notation as complex as the grammars.
3. Regular expressions generally provide a more concise and easier-to-understand notation for tokens than grammars.

4. More efficient lexical analyzers can be constructed automatically from regular expressions than from arbitrary grammars.

- There are no firm guidelines as to what to put into the lexical rules, as opposed to the syntactic rules.
- Regular expressions are useful to describe constructs such as identifiers, numbers...
- Grammars are most useful for describing nested structures such as balanced parentheses, corresponding if-then-else...
1 Introduction

2 Context-free grammar
   - Definition and notation
   - Derivations and Parse Tree
   - Ambiguity of a grammar
   - Verifying the language supported by a grammar
   - Context-free grammar and regular expression
   - Guidelines for writing a grammar
     - General guidelines
     - Eliminating the ambiguity
     - Eliminating the left recursion
     - Left factoring

3 Parsing with a grammar

4 Generate a syntactic parser with Yacc or JavaCC

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
An ambiguous grammar can be rewritten to eliminate the ambiguity.

**Grammar:**

\[
\text{statement} \rightarrow \text{if expression then statement} \\
| \text{if expression then statement else statement} \\
| \text{other}
\]

**Input:** \(\text{if } E_1 \text{ then if } E_2 \text{ else } S_1 \text{ else } S_2\)

The first tree is preferred according to “Match each else with the closest unmatched then.” This rule is rarely built into productions.
The disambiguation of this “if-then-else” problem may be included into a new grammar.

\[
\begin{align*}
\text{statement} & \rightarrow \text{if expression then statement} \\
& \quad | \text{if expression then statement else statement} \\
& \quad | \text{other}
\end{align*}
\]

\[
\begin{align*}
\text{statement} & \rightarrow \text{matched_statement} \\
& \quad | \text{open_statement} \\
\text{matched_statement} & \rightarrow \text{if expression then matched_statement else matched_statement} \\
& \quad | \text{other} \\
\text{open_statement} & \rightarrow \text{if expression then statement} \\
& \quad | \text{if expression then matched_statement else open_statement}
\end{align*}
\]
1 Introduction

2 Context-free grammar
   - Definition and notation
   - Derivations and Parse Tree
   - Ambiguity of a grammar
   - Verifying the language supported by a grammar
   - Context-free grammar and regular expression
   - Guidelines for writing a grammar
     - General guidelines
     - Eliminating the ambiguity
     - Eliminating the left recursion
     - Left factoring

3 Parsing with a grammar

4 Generate a syntactic parser with Yacc or JavaCC
Definition (Left-Recursive Grammar)

A grammar is left recursive if it has a nonterminal $A$ such that there is a derivation $A \xrightarrow{+} A\alpha$ for some string $\alpha$.

Top-down parsing methods cannot handle left-recursive grammars.

- A transformation is needed to eliminate left recursion.
- The algorithm to systematically eliminates left recursion from a grammar if the grammar has no cycle nor $\epsilon$-production:
  - $\neg (A \xrightarrow{+} A)$ — no cycle
  - $\neg (A \rightarrow \epsilon)$ — no $\epsilon$-production
Eliminating Left Recursion

**Input**: Grammar $G$.

**Output**: An equivalent grammar with no left recursion.

```plaintext
begin
  while $\exists A | (A \rightarrow A \gamma) \in G$ do
    foreach $p = (A \rightarrow b \delta) \in G | b \neq A$ do
      $G \leftarrow G \setminus \{p\}$;
      $G \leftarrow G \cup \{(R_A \rightarrow b \delta R_A)\}$;
    end
    $G = G \cup \{(R_A \rightarrow \epsilon)\}$;
  end
  foreach $p = (A \rightarrow A \omega) \in G$ do
    $G \leftarrow G \setminus \{p\}$;
    $G \leftarrow G \cup \{(A \rightarrow \omega R_A)\}$;
  end
end
```

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
Example of Eliminating Left Recursion

\[
E \rightarrow E + E \\
\mid E \ast E \\
\mid - E \\
\mid ( E ) \\
\mid \text{id}
\]

\[
E \rightarrow - E R_E \\
\mid ( E ) R_E \\
\mid \text{id} R_E \\
\mid + E R_E \\
\mid * E R_E \\
\mid \epsilon
\]
Outline

1 Introduction

2 Context-free grammar
   ■ Definition and notation
   ■ Derivations and Parse Tree
   ■ Ambiguity of a grammar
   ■ Verifying the language supported by a grammar
   ■ Context-free grammar and regular expression
   ■ Guidelines for writing a grammar
      ■ General guidelines
      ■ Eliminating the ambiguity
      ■ Eliminating the left recursion
      ■ Left factoring

3 Parsing with a grammar

4 Generate a syntactic parser with Yacc or JavaCC

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
When the choice between two alternatives $A$-productions is not clear, we may be able to rewrite the productions to defer the decision until enough of the input has been seen that we can make the right choice.

\[
\text{statement} \quad \rightarrow \quad \text{if } \text{expression} \text{ then } \text{statement} \text{ else } \text{statement} \\
\quad | \quad \text{if } \text{expression} \text{ then } \text{statement} \\
\quad | \quad \text{other}
\]

Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive, or top-down, parsing.
Example of Left Factoring

\[
\text{statement} \rightarrow \begin{align*}
\text{if expression then statement else statement} \\
\text{if expression then statement} \\
\text{other}
\end{align*}
\]

\[
\text{statement} \rightarrow \begin{align*}
\text{if expression then statement else statement} \\
\text{other}
\end{align*}
\]

\[
\text{else_statement} \rightarrow \begin{align*}
\text{else statement} \\
\epsilon
\end{align*}
\]
Algorithm for Left Factoring

**Input**: Grammar $G$.

**Output**: An equivalent left-factored grammar.

```plaintext
begin
  while $\exists A \in G | (A \to \alpha \gamma), (A \to \alpha \delta)$ do
    foreach $p = (A \to \alpha \omega) \in G$ do
      $G \leftarrow G \setminus \{p\}$;
      if $\omega \neq \epsilon$ then
        $G \leftarrow G \cup \{(R_A \to \omega)\}$;
      end
    end
  end
  $G \leftarrow G \cup \{(A \to \alpha R_A)\}$;
  $G \leftarrow G \cup \{(R_A \to \epsilon)\}$;
end
```

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
Outline

1. Introduction
2. Context-free grammar
3. Parsing with a grammar
   - Top-down parsing
   - Bottom-up parsing
   - LR parsing
4. Generate a syntactic parser with Yacc or JavaCC
5. Conclusion
Outline

1 Introduction

2 Context-free grammar

3 Parsing with a grammar
   - Top-down parsing
     - Principles
     - Recursive-descent parsing
     - FIRST and FOLLOW
     - $LL(1)$ grammars
     - Nonrecursive predictive parsing
     - Error recovery in predictive parsing
   - Bottom-up parsing
     - $LR$ parsing

4 Generate a syntactic parser with Yacc or JavaCC

5 Conclusion
Top-down parsing can be viewed as the problem of constructing a parse tree for the input string, starting from the root and creating the nodes of the parse tree in preorder.

Top-down parsing can be viewed as finding a leftmost derivation for an input string.

The rest of this section uses the following grammar as illustration; and uses the input string \( \text{id} + \text{id} \ast \text{id} \).

\[
\begin{align*}
E & \rightarrow T E' \\
E' & \rightarrow + T E' \\
& \mid \epsilon \\
T & \rightarrow F T' \\
T' & \rightarrow * F T' \\
& \mid \epsilon \\
F & \rightarrow ( E ) \\
& \mid \text{id}
\end{align*}
\]
Several top-down parsing methods exist, the two majors are:

- **Recursive-descent parsing**: a general form, which may require backtracking to find the correct $A$-production to be applied.

- **Predictive parsing**: a special case of recursive-descent parsing, where no backtracking is required. The $A$-production is chosen by looking ahead at the input a fixed number of symbols.

The class of grammars dedicated to the predictive parsers looking $k$ symbols ahead in the input is called $LL(k)$ class.
1 Introduction

2 Context-free grammar

3 Parsing with a grammar
   - Top-down parsing
     - Principles
     - Recursive-descent parsing
     - FIRST and FOLLOW
     - $LL(1)$ grammars
     - Nonrecursive predictive parsing
     - Error recovery in predictive parsing
   - Bottom-up parsing
   - $LR$ parsing

4 Generate a syntactic parser with Yacc or JavaCC

5 Conclusion
A recursive-descent parsing program consists of a set of procedures, one for each nonterminal.

Execution begins with the procedure for the start symbol.

The pseudo-code for each nonterminal is:

```
Procedure A
Input : A production A → α₁...αₖ.
begin
  for i ← 1 to k do
    if αᵢ is a nonterminal then
      call αᵢ();
    else if αᵢ = current input symbol a then
      forward ← forward + 1 // Move input pointer
    else
      Report an error;
  end
end
```
A left-recursive grammar can cause a recursive-descent parser to go into an infinite loop.

That is, when we try to expand a nonterminal $A$, we may eventually find ourselves again trying to expand $A$ without having consumed input.
1 Introduction

2 Context-free grammar

3 Parsing with a grammar
   - Top-down parsing
     - Principles
     - Recursive-descent parsing
     - FIRST and FOLLOW
     - $LL(1)$ grammars
     - Nonrecursive predictive parsing
     - Error recovery in predictive parsing
   - Bottom-up parsing
     - $LR$ parsing

4 Generate a syntactic parser with Yacc or JavaCC

5 Conclusion

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
The construction of both top-down and bottom-up parsers is aided by two functions associated with a grammar G:

1. FIRST
2. FOLLOW

These functions allow us to choose which production to apply, based on the next input symbol.

During panic-mode error recovery the set of tokens replied by FOLLOW can be used as synchronizing tokens.
Define $FIRST(\alpha)$, where $\alpha$ is any string of grammar symbols, to be the set of terminals that begin strings derived from $\alpha$.

If $\alpha \Rightarrow^* \epsilon$, then $\epsilon$ is also in $FIRST(\alpha)$.

**Example**

$A \Rightarrow^* c \gamma$

$FIRST(A) = \{c\}$
To compute $\text{FIRST}(X)$ for a grammar symbol $X$, apply the following rules until no more terminals or $\epsilon$ can be added to any FIRST set.

1. If $X$ is a terminal, then $\text{FIRST}(X) = \{X\}$.
2. If $X$ is a nonterminal and $X \rightarrow Y_1 Y_2 \ldots Y_k$ is a production for some $k \geq 1$, then place $a$ in $\text{FIRST}(X)$ if for some $i$, $a$ is in $\text{FIRST}(Y_i)$, and $\epsilon$ is in all of $\text{FIRST}(Y_1), \ldots, \text{FIRST}(Y_{k-1})$; that is, $Y_1 Y_2 \ldots Y_k \Rightarrow^* \epsilon$. If $\epsilon$ is in $\text{FIRST}(Y_j)$ for all $j \in \{1, 2, \ldots, k\}$, then add $\epsilon$ to $\text{FIRST}(X)$.
3. If $X \rightarrow \epsilon$ is a production, then add $\epsilon$ to $\text{FIRST}(X)$.

Add to $\text{FIRST}(X_1 X_2 \ldots X_n)$ all non-$\epsilon$ symbols of $\text{FIRST}(X_i)$ for $i \in \{1 \ldots n\}$. 
Define FOLLOW(A), where A is a nonterminal, to be the set of terminals a that can appear immediately to the right of A in some sentential form.

The set of terminals a such that there exists a derivation of the form $S \Rightarrow^{*} \alpha \ A \ a \ \beta$, for some $\alpha$ and $\beta$.

Note that there may have been symbols between $A$ and $a$, at some time during the derivation, but if so, they derive $\epsilon$ and disappeared.

If $A$ can be the rightmost symbol, then $\text{eof}$ (or usually $\$$) is in FOLLOW(A).

Example

$A \Rightarrow^* c \gamma$

$a \in \text{FOLLOW}(A)$
To compute \( \text{FOLLOW}(A) \) for a nonterminal \( A \), apply the following rules until nothing can be added to any \( \text{FOLLOW} \) set.

1. Place \texttt{eof} in \( \text{FOLLOW}(S) \), where \( S \) is the start symbol, and \texttt{eof} is the input right endmarker.

2. If there is a production \( A \rightarrow \alpha B \beta \), then everything in \( \text{FIRST}(b) \), except \( \epsilon \) is in \( \text{FOLLOW}(B) \).

3. If there is a production \( A \rightarrow \alpha B \), or a production \( A \rightarrow \alpha B \beta \), where \( \text{FIRST}(\beta) \) contains \( \epsilon \), then everything in \( \text{FOLLOW}(A) \) is in \( \text{FOLLOW}(B) \).
1 Introduction

2 Context-free grammar

3 Parsing with a grammar
   - Top-down parsing
     - Principles
     - Recursive-descent parsing
     - FIRST and FOLLOW
     - \textit{LL}(1) grammars
     - Nonrecursive predictive parsing
     - Error recovery in predictive parsing
   - Bottom-up parsing
     - \textit{LR} parsing

4 Generate a syntactic parser with Yacc or JavaCC

5 Conclusion

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
Predictive parsers, that is, recursive-descent parsers needing no backtracking, can be constructed for a class of grammars called $LL(1)$.

- Left-to-right input scanning,
- Leftmost derivation,
- 1 input symbol is used for lookahead to make parsing action decisions.

The class of $LL(1)$ grammars is rich enough to cover most programming constructs, although care is needed in writing a suitable grammar for the source language eg., no left-recursive nor ambiguous grammar can be $LL(1)$.
Definition of a $LL(1)$ Grammar

A grammar $G$ is $LL(1)$ iff whenever $A \rightarrow \alpha \mid \beta$ are two distinct productions of $G$, the following conditions hold:

1. For nonterminal $a$ do both $\alpha$ and $\beta$ derive strings beginning with $a$.

2. At most one of $\alpha$ and $\beta$ can derive the empty string.

3. If $\beta \Rightarrow^* \epsilon$, then $\alpha$ does not derive any string beginning with a terminal in FOLLOW($A$). Likewise, if $\alpha \Rightarrow^* \epsilon$, then $\beta$ does not derive any string beginning with a terminal in FOLLOW($A$).

| $statement_list$    | $\rightarrow$ | $statement$  $statement_list$
|---------------------|---------------|--------------|
|                     | $\mid$        | $\epsilon$

| $statement$         | $\rightarrow$ | $if$ ( $expression$ ) $statement$ $else$ $statement$
|---------------------|---------------|--------------|
|                     | $\mid$        | $while$ ( $expression$ ) $statement$
|                     | $\mid$        | $\{ $ $statement_list$ $\}$
To parse an input string, a table should be build. It permits to determine the production to use from a given production and the input symbol.

The following algorithm permits to collects informations from FIRST and FOLLOW sets into a predictive parsing table $M[A, a]$, where $A$ is a nonterminal, and $a$ is a terminal or $\text{eof}$.
The algorithm is based on the idea:

1. The production \( A \to \alpha \) is chosen if the next input symbol \( a \in \text{FIRST}(\alpha) \).

2. When \( \alpha \Rightarrow^* \epsilon \), we should again choose \( A \to \alpha \), if the current input symbol is in \( \text{FOLLOW}(A) \), or if the \text{eof} on the input has been reached and \text{eof} is in \( \text{FOLLOW}(A) \).

If a cell of the predictive parsing table contains more than one information, then the grammar associated to the table is ambiguous.
Algorithm for Building the Predictive-Parsing Table (#2)

- **INPUT:** Grammar $G$.
- **OUTPUT:** Parsing table $M$.
- **METHOD:** For each production $A \rightarrow \alpha$ of the grammar, do the following:
  - For each terminal $a$ in $\text{FIRST}(\alpha)$, add $A \rightarrow \alpha$ to $M[A, a]$.
  - If $\epsilon$ is in $\text{FIRST}(\alpha)$, then for each terminal $b$ in $\text{FOLLOW}(A)$, add $A \rightarrow \alpha$ to $M[A, b]$.
  - If $\epsilon$ is in $\text{FIRST}(\alpha)$ and $\text{eof}$ is in $\text{FOLLOW}(A)$, add $A \rightarrow \alpha$ to $M[A, \text{eof}]$ as well.

If, after performing the above, there is no production in $M[A, a]$, then set $M[A, a]$ to error (generally replaced by an empty string in the table).
### Example of Table Building

#### For production: \((1)\) \(E \rightarrow T E'\)

FIRST(\(TE'\)) = FIRST(T) = FIRST(F) = \{(:, id}\}

<table>
<thead>
<tr>
<th></th>
<th>(id)</th>
<th>(+)</th>
<th>(*)</th>
<th>( )</th>
<th>eof</th>
</tr>
</thead>
<tbody>
<tr>
<td>(E)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(E')</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(T)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(T')</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(F)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example of Table Building

For production: \( E \rightarrow TE' \)
FIRST(TE') = FIRST(T) = FIRST(F) = \{ (, \text{id}) \}
Then put the production in \( M[E,()] \) and \( M[E,\text{id}] \); the rest of the line is error

<table>
<thead>
<tr>
<th></th>
<th>id</th>
<th></th>
<th></th>
<th></th>
<th>ε</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>E</td>
<td>( T )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(2)</td>
<td>E'</td>
<td></td>
<td>+</td>
<td>T</td>
<td>E'</td>
</tr>
<tr>
<td>(3)</td>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td>( F ) T'</td>
</tr>
<tr>
<td>(4)</td>
<td>T'</td>
<td>*</td>
<td></td>
<td>F</td>
<td>T'</td>
</tr>
<tr>
<td>(5)</td>
<td>F</td>
<td>(</td>
<td></td>
<td>E</td>
<td>)</td>
</tr>
<tr>
<td>(6)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(7)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(8)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>\text{id}</td>
</tr>
</tbody>
</table>
Example of Table Building

For production: \((2)\) \(E' \rightarrow + T E'\)

FIRST\((+ TE')\) = \{+\}

Then put the production in \(M[E', +]\)

\[
\begin{array}{|c|}
\hline
(1) & E \rightarrow TE' \\
(2) & E' \rightarrow + TE' \\
(3) & T \rightarrow F T' \\
(4) & T' \rightarrow * F T' \\
(5) & F \rightarrow (E) \\
(6) & | \epsilon \\
(7) & | id \\
(8) & | \epsilon \\
\hline
\end{array}
\]

<table>
<thead>
<tr>
<th>id</th>
<th>+</th>
<th>*</th>
<th>(   )</th>
<th>eof</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>(1)</td>
<td></td>
<td></td>
<td>(1)</td>
</tr>
<tr>
<td>E'</td>
<td></td>
<td>(2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T'</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Example of Table Building

**For production:** (3) \( E' \rightarrow \epsilon \)

- FIRST(\( \epsilon \)) = \( \{ \epsilon \} \)
- FOLLOW(\( E' \)) = \( \{ \), eof \} 

Put the production in \( M[E', ] \) (rule 2)

<table>
<thead>
<tr>
<th>( E )</th>
<th>( T )</th>
<th>( \epsilon )</th>
<th>( ( )</th>
<th>( ) )</th>
<th>( {\text{eof}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>+</td>
<td>*</td>
<td>(</td>
<td>)</td>
<td>eof</td>
</tr>
<tr>
<td>( E )</td>
<td>( (1) )</td>
<td>( (1) )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( E' )</td>
<td>( (2) )</td>
<td>( (3) )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( T )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( T' )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( F )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example of Table Building

For production: \( (3) \ E' \rightarrow \epsilon \)

FIRST(\( \epsilon \)) = \{\( \epsilon \)\}

FOLLOW(\( E' \)) = \{, \text{eof}\}

Put the production in \( M[E', ] \) (rule 2)

Put the production in \( M[E', \text{eof}] \) (rule 3)

<table>
<thead>
<tr>
<th>id</th>
<th>+</th>
<th>*</th>
<th>(   )</th>
<th>eof</th>
</tr>
</thead>
<tbody>
<tr>
<td>( E )</td>
<td>(1)</td>
<td></td>
<td>(1)</td>
<td></td>
</tr>
<tr>
<td>( E' )</td>
<td>(2)</td>
<td></td>
<td>(3)</td>
<td>(3)</td>
</tr>
<tr>
<td>( T )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( T' )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( F )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example of Table Building

For production: (4) $T \rightarrow F T'$
FIRST($F T'$) = FIRST($F$) = \{[, \text{id}]\}
Put the production in $M[T, []$ and $M[T, \text{id}]$

<table>
<thead>
<tr>
<th>Production</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E \rightarrow TE'$</td>
<td>id</td>
<td>+</td>
<td>*</td>
<td>(</td>
<td>)</td>
<td>eof</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$E' \rightarrow + TE'$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$T \rightarrow F T'$</td>
<td></td>
<td></td>
<td>(2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$T' \rightarrow * FT'$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(4)</td>
<td></td>
</tr>
<tr>
<td>$F \rightarrow (E)$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
### Example of Table Building

For production: \((5)\) \(T' \rightarrow * \ F \ T'\)

FIRST(* \ F \ T') = \{ * \}

Put the production in \(M[T', \ast]\)

<table>
<thead>
<tr>
<th></th>
<th>id</th>
<th>+</th>
<th>*</th>
<th>(</th>
<th>)</th>
<th>eof</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>(1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(1)</td>
</tr>
<tr>
<td>E'</td>
<td></td>
<td>(2)</td>
<td></td>
<td></td>
<td></td>
<td>(3)</td>
</tr>
<tr>
<td>T</td>
<td>(4)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(4)</td>
</tr>
<tr>
<td>T'</td>
<td></td>
<td></td>
<td>(5)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example of Table Building

For production: (6) $T' \rightarrow \epsilon$

FIRST($\epsilon$) = \{\epsilon\}

FOLLOW($T'$) = \{+,\), eof\}

Put the production in $M[T', +]$ and $M[T, )$ (rule 2)

Put the production in $M[T', \text{eof}]$ (rule 3)

<table>
<thead>
<tr>
<th>id</th>
<th>+</th>
<th>*</th>
<th>(</th>
<th>)</th>
<th>eof</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E$</td>
<td>(1)</td>
<td></td>
<td></td>
<td>(1)</td>
<td></td>
</tr>
<tr>
<td>$E'$</td>
<td>(2)</td>
<td></td>
<td></td>
<td></td>
<td>(3)</td>
</tr>
<tr>
<td>$T$</td>
<td>(4)</td>
<td></td>
<td></td>
<td></td>
<td>(4)</td>
</tr>
<tr>
<td>$T'$</td>
<td>(6)</td>
<td>(5)</td>
<td></td>
<td>(6)</td>
<td>(6)</td>
</tr>
<tr>
<td>$F$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Example of Table Building

**For production:** $(7) \ F \rightarrow (\ E)\$

$\text{FIRST}((\ E)) = \{()\}$

Put the production in $M[F,()]$

<table>
<thead>
<tr>
<th>id</th>
<th>$+$</th>
<th>$*$</th>
<th>(</th>
<th>)</th>
<th>eof</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E$</td>
<td>$(1)$</td>
<td></td>
<td>$(1)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$E'$</td>
<td></td>
<td>$(2)$</td>
<td></td>
<td>$(3)$</td>
<td>$(3)$</td>
</tr>
<tr>
<td>$T$</td>
<td>$(4)$</td>
<td></td>
<td>$(4)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$T'$</td>
<td>$(6)$</td>
<td>$(5)$</td>
<td>$(6)$</td>
<td>$(6)$</td>
<td></td>
</tr>
<tr>
<td>$F$</td>
<td></td>
<td></td>
<td></td>
<td>$(7)$</td>
<td></td>
</tr>
</tbody>
</table>

© 2012–2014 Stéphane Galland

stephane.galland@utbm.fr – http://www.multiagent.fr
### Example of Table Building

<table>
<thead>
<tr>
<th>Production</th>
<th>Symbol</th>
<th>FIRST(id)</th>
<th>Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>For production: $(9) F \rightarrow \text{id}$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FIRST(id) = ${\text{id}}$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Put the production in $M[F, \text{id}]$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Production</th>
<th>Symbol</th>
<th>FIRST(id)</th>
<th>Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(1) E \rightarrow T E'$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$(2) E' \rightarrow + T E'$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$(3) T \rightarrow F T'$</td>
<td>$\epsilon$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$(4) T' \rightarrow * F T'$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$(5) F \rightarrow ( E )$</td>
<td>$\text{id}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$(6) T' \rightarrow * F T'$</td>
<td>$\epsilon$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$(7) F \rightarrow ( E )$</td>
<td>$\text{id}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$(8) F \rightarrow ( E )$</td>
<td>$\text{id}$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E$</td>
<td>$(1)$</td>
</tr>
<tr>
<td>$E'$</td>
<td>$(2)$</td>
</tr>
<tr>
<td>$T$</td>
<td>$(4)$</td>
</tr>
<tr>
<td>$T'$</td>
<td>$(6)$</td>
</tr>
<tr>
<td>$F$</td>
<td>$(8)$</td>
</tr>
<tr>
<td>$+$</td>
<td>$(1)$</td>
</tr>
<tr>
<td>$*$</td>
<td>$(3)$</td>
</tr>
<tr>
<td>$( )$</td>
<td>$(3)$</td>
</tr>
<tr>
<td>$\text{id}$</td>
<td>$(6)$</td>
</tr>
<tr>
<td>$\text{id}$</td>
<td>$(7)$</td>
</tr>
<tr>
<td>$\text{eof}$</td>
<td></td>
</tr>
</tbody>
</table>
Outline

1 Introduction

2 Context-free grammar

3 Parsing with a grammar
   - Top-down parsing
     - Principles
     - Recursive-descent parsing
     - FIRST and FOLLOW
     - $LL(1)$ grammars
     - Nonrecursive predictive parsing
     - Error recovery in predictive parsing
   - Bottom-up parsing
     - $LR$ parsing

4 Generate a syntactic parser with Yacc or JavaCC

5 Conclusion

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
A nonrecursive predictive parser can be built by maintaining a stack explicitly, rather than implicitly via recursive calls.

The parser mimics a leftmost derivation. If $w$ is the input that has been matched so far, then the stack holds a sequence of grammar symbols $a$ such that:

$$S \Rightarrow^* w \alpha$$

The table-driven parser has an input buffer, a stack containing a sequence of grammar symbols, a parsing table, and an output stream.
Algorithm of the Nonrecursive Predictive Parsing

Input : A string $w$, a parsing table $M$ for grammar $G$, and a start symbol $s_0$.
Output : If $w$ is in $L(G)$, a leftmost derivation of $w$; otherwise, an error indication.

begin
  $a \leftarrow \text{inputSymbol}();$
  push $(S, \text{eof})$;
  push $(S, s_0)$;
  while $\text{topOf}(S) \neq \text{eof}$ do
    $X \leftarrow \text{topOf}(S)$;
    if $X = a$ then
      pop $(S)$; $a \leftarrow \text{nextInputSymbol}();$
    else if $X$ is a terminal then
      Report an error;
    else if $M[X, a] = X \rightarrow Y_1 \ldots Y_n$ then
      print $(X \rightarrow Y_1 \ldots Y_n)$;
      pop $(S)$;
      for $i \leftarrow n$ to 1 do push $(S, Y_i)$;
    else
      Report an error
  end
end
Example of Nonrecursive Predictive Parsing

```
begin
  a ← inputSymbol ();
push (S, eof);
push (S, s0);
while top0f(S) ≠ eof do
  X ← top0f (S);
  if X = a then
    pop (S); a ← nextInputSymbol ();
  else if X is a terminal then
    Report an error;
  else if M[X, a] = X → Y1 ... Yn then
    print (X → Y1 ... Yn);
    pop (S);
    for i ← n to 1 do push (S, Yi);
  else
    Report an error;
end
end
```

```
E → T E'  (1)
E' → + T E'  (2)
T → F T'  (3)
T' → * F T'  (4)
F → ( E )  (5)
F → id  (6)
```

Stack: 
```
  |  |  |  |  |  |  |  |
```

Input: 
```
id + id * id eof
```

Output: 
```
Example of Nonrecursive Predictive Parsing

begin
a ← inputSymbol ();
push (S, eof);
push (S, s0);
while top0f(S) ≠ eof do
    X ← top0f (S);
    if X = a then
        pop (S); a ← nextInputSymbol ();
    else if X is a terminal then
        Report an error;
    else if M[X, a] = X → Y1 ... Yn then
        print (X → Y1 ... Yn);
        pop (S);
        for i ← n to 1 do push (S, Yi) ;
    else
        Report an error;
end
end

```
begin
a ← inputSymbol ();
push (S, eof);
push (S, s0);
while top0f(S) ≠ eof do
    X ← top0f (S);
    if X = a then
        pop (S); a ← nextInputSymbol ();
    else if X is a terminal then
        Report an error;
    else if M[X, a] = X → Y1 ... Yn then
        print (X → Y1 ... Yn);
        pop (S);
        for i ← n to 1 do push (S, Yi) ;
    else
        Report an error;
end
end
```

### Context-free Grammar

<table>
<thead>
<tr>
<th></th>
<th>id</th>
<th>+</th>
<th>*</th>
<th>( )</th>
<th>eof</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>1</td>
<td></td>
<td></td>
<td>(1)</td>
<td></td>
</tr>
<tr>
<td>E'</td>
<td>2</td>
<td></td>
<td></td>
<td>(1)</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>4</td>
<td></td>
<td></td>
<td>(3)</td>
<td>(3)</td>
</tr>
<tr>
<td>T'</td>
<td>6</td>
<td>5</td>
<td></td>
<td>(6)</td>
<td>(6)</td>
</tr>
<tr>
<td>F</td>
<td>8</td>
<td></td>
<td></td>
<td>(7)</td>
<td></td>
</tr>
</tbody>
</table>

### Example Grammar

<table>
<thead>
<tr>
<th></th>
<th>E</th>
<th>→</th>
<th>T E'</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>E'</td>
<td>→</td>
<td>+ T E'</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>→</td>
<td>F T'</td>
</tr>
<tr>
<td>3</td>
<td>T'</td>
<td>→</td>
<td>* F T'</td>
</tr>
<tr>
<td>4</td>
<td>F</td>
<td>→</td>
<td>( E )</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>→</td>
<td>id</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>→</td>
<td>ε</td>
</tr>
</tbody>
</table>

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
Example of Nonrecursive Predictive Parsing

begin
  a ← inputSymbol();
push (S, eof);
push (S, s0);
while topOf(S) ≠ eof do
  X ← topOf(S);
  if X = a then
    pop (S); a ← nextInputSymbol();
  else if X is a terminal then
    Report an error;
  else if M[X, a] = X → Y₁ ... Yₙ then
    print (X → Y₁ ... Yₙ);
    pop (S);
    for i ← n to 1 do push (S, Yᵢ);
  else
    Report an error;
end
end

<table>
<thead>
<tr>
<th></th>
<th>id</th>
<th>+</th>
<th>*</th>
<th>(</th>
<th>)</th>
<th>eof</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>(1)</td>
<td></td>
<td></td>
<td>(1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E'</td>
<td>(2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>(4)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T'</td>
<td>(6)</td>
<td>(5)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>(8)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

stack: input: output:

id + id * id eof

(1) E → T E'
(2) E' → + T E'
(3) | ε
(4) T → F T'
(5) T' → * F T'
(6) | ε
(7) F → ( E )
(8) | id
Example of Nonrecursive Predictive Parsing

begin
  a ← inputSymbol();
  push (S, eof);
  push (S, sq);
  while topOf(S) ≠ eof do
    X ← topOf(S);
    if X = a then
      pop (S); a ← nextInputSymbol();
    else if X is a terminal then
      Report an error;
    else if M[X, a] = X → Y₁ ... Yₙ then
      print (X → Y₁ ... Yₙ);
      pop (S);
      for i ← n to 1 do push (S, Yᵢ);
    else
      Report an error;
  end
end

stack: input: output:

<table>
<thead>
<tr>
<th>(1)</th>
<th>(1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2)</td>
<td>(3)</td>
</tr>
<tr>
<td>(4)</td>
<td>(4)</td>
</tr>
<tr>
<td>(6)</td>
<td>(6)</td>
</tr>
<tr>
<td>(7)</td>
<td>(7)</td>
</tr>
</tbody>
</table>

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
Example of Nonrecursive Predictive Parsing

begin
  a ← inputSymbol();
  push (S, eof);
  push (S, s0);
while top0f(S) ≠ eof do
  X ← top0f(S);
  if X = a then
      pop (S); a ← nextInputSymbol();
  else if X is a terminal then
      Report an error;
  else if M[X, a] = X → Y₁...Yₙ then
      print (X → Y₁...Yₙ);
      pop (S);
      for i ← n to 1 do push (S, Yᵢ);
  else
      Report an error;
end
end

<table>
<thead>
<tr>
<th></th>
<th>id</th>
<th>+</th>
<th>*</th>
<th>( )</th>
<th>eof</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>(1)</td>
<td></td>
<td></td>
<td>(1)</td>
<td></td>
</tr>
<tr>
<td>E’</td>
<td>(2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>(4)</td>
<td></td>
<td></td>
<td>(3)</td>
<td>(3)</td>
</tr>
<tr>
<td>T’</td>
<td>(6)</td>
<td>(5)</td>
<td></td>
<td>(6)</td>
<td>(6)</td>
</tr>
<tr>
<td>F</td>
<td>(8)</td>
<td></td>
<td></td>
<td>(7)</td>
<td></td>
</tr>
</tbody>
</table>

input: id + id * id eof

output:

E → T E’

E → T  
E’ → + T E’
T → F T’
T’ → * F T’
F → ( E )
F → id

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
Example of Nonrecursive Predictive Parsing

```
begin
    a ← inputSymbol();
push (S, eof);
push (S, s0);
while top0f(S) ≠ eof do
    X ← top0f(S);
    if X = a then
        pop (S); a ← nextInputSymbol();
    else if X is a terminal then
        Report an error;
    else if M[X, a] = X → Y1 . . . Yn then
        print (X → Y1 . . . Yn);
        pop (S);
        for i ← n to 1 do push (S, Yi);
    else
        Report an error;
end
end
```

Stack:

- `id + id * id eof`

Input:
- `id + id * id eof`

Output:
- `E → T E'`

Context-free grammar:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Production</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>(1)</td>
</tr>
<tr>
<td>E'</td>
<td>(2)</td>
</tr>
<tr>
<td>T</td>
<td>(3)</td>
</tr>
<tr>
<td>T'</td>
<td>(4)</td>
</tr>
<tr>
<td>F</td>
<td>(5)</td>
</tr>
<tr>
<td>id</td>
<td>(6)</td>
</tr>
<tr>
<td>+</td>
<td>(7)</td>
</tr>
</tbody>
</table>

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
Example of Nonrecursive Predictive Parsing

```
begin
  a ← inputSymbol();
push (S, eof);
push (S, s0);
while top0f(S) ≠ eof do
  X ← top0f(S);
  if X = a then
    pop (S); a ← nextInputSymbol();
  else if X is a terminal then
    Report an error;
  else if M[X, a] = X → Y₁ ... Yₙ then
    print (X → Y₁ ... Yₙ);
pop (S);
  for i ← n to 1 do push (S, Yᵢ);
  else
    Report an error;
end
end
```

```
stack:

input:
  id + id * id eof

output:

E → T E'
T → F T'
F → ( E )
```

```
<table>
<thead>
<tr>
<th></th>
<th>id</th>
<th>+</th>
<th>*</th>
<th>( )</th>
<th>eof</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>(1)</td>
<td></td>
<td></td>
<td>(1)</td>
<td></td>
</tr>
<tr>
<td>E'</td>
<td>(2)</td>
<td></td>
<td></td>
<td>(3)</td>
<td>(3)</td>
</tr>
<tr>
<td>T</td>
<td>(4)</td>
<td></td>
<td></td>
<td>(4)</td>
<td></td>
</tr>
<tr>
<td>T'</td>
<td>(6)</td>
<td></td>
<td></td>
<td>(6)</td>
<td>(6)</td>
</tr>
<tr>
<td>F</td>
<td>(8)</td>
<td></td>
<td></td>
<td>(7)</td>
<td></td>
</tr>
</tbody>
</table>
```

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
Example of Nonrecursive Predictive Parsing

begin
\[ a \leftarrow \text{inputSymbol}() ; \]
\[ \text{push} (S, \text{eof}) ; \]
\[ \text{push} (S, s_0) ; \]
\[ \text{while} \top(S) \neq \text{eof} \text{ do} \]
\[ X \leftarrow \top(S) ; \]
\[ \text{if } X = a \text{ then} \]
\[ \text{pop} (S) ; a \leftarrow \text{nextInputSymbol}() ; \]
\[ \text{else if } X \text{ is a terminal then} \]
\[ \text{Report an error;} \]
\[ \text{else if } M[X, a] = X \rightarrow Y_1 \ldots Y_n \text{ then} \]
\[ \text{print}(X \rightarrow Y_1 \ldots Y_n) ; \]
\[ \text{pop} (S) ; \]
\[ \text{for } i \leftarrow n \text{ to } 1 \text{ do } \text{push} (S, Y_i) ; \]
\[ \text{else} \]
\[ \text{Report an error;} \]
end
\end{verbatim}

**Stack:**
- id
- +
- *
- ( )
- \text{eof}

**Input:**
- id
- +
- *
- id
- ( )
- \text{eof}

**Output:**
- \( E \rightarrow T E' \)

**Production Rules:**

<table>
<thead>
<tr>
<th>Rule</th>
<th>Right Side</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( E \rightarrow T E' )</td>
</tr>
<tr>
<td>2</td>
<td>( E' \rightarrow + T E' )</td>
</tr>
<tr>
<td>3</td>
<td>( \epsilon )</td>
</tr>
<tr>
<td>4</td>
<td>( T \rightarrow F T' )</td>
</tr>
<tr>
<td>5</td>
<td>( T' \rightarrow * F T' )</td>
</tr>
<tr>
<td>6</td>
<td>( \epsilon )</td>
</tr>
<tr>
<td>7</td>
<td>( F \rightarrow ( E ) )</td>
</tr>
<tr>
<td>8</td>
<td>( \text{id} )</td>
</tr>
</tbody>
</table>

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
Example of Nonrecursive Predictive Parsing

begin
    a ← inputSymbol ();
    push ( S, eof );
    push ( S, s0 );
    while topOf ( S ) ≠ eof do
        X ← topOf ( S );
        if X = a then
            pop ( S ); a ← nextInputSymbol ();
        else if X is a terminal then
            Report an error;
        else if M [ X, a ] = X → Y₁ ... Yₙ then
            print ( X → Y₁ ... Yₙ );
            pop ( S );
            for i ← 1 to n do push ( S, Yᵢ );
        else
            Report an error;
    end
end

stack: input: output:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>+</td>
<td>*</td>
<td>(</td>
<td>)</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>E</td>
<td>(1)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E'</td>
<td>(2)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>(4)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T'</td>
<td>(6)</td>
<td>(5)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>(8)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
Example of Nonrecursive Predictive Parsing

begin
    a ← inputSymbol ();
    push (S, eof);
    push (S, s0);
    while top0f(S) ≠ eof do
        X ← top0f (S);
        if X = a then
            pop (S); a ← nextInputSymbol ();
        else if X is a terminal then
            Report an error;
        else if M[X, a] = X → Y1 ... Yn then
            print (X → Y1 ... Yn);
            pop (S);
            for i ← n to 1 do push (S, Yi) ;
        else
            Report an error;
    end
end

stack:    input:    output:
  id + id * id eof

  a

E    T    E'
E    +    T    E'
T    ( )
T'    *    T    T'
F    ( )

(1) E → T E'
(2) E' → + T E'
(3) T → F T'
(4) T' → * F T'
(5) F → ( E )
(6) | ε
(7) | id

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
Example of Nonrecursive Predictive Parsing

begin
  a ← inputSymbol ()
  push (S, eof)
  push (S, s₀)
  while topOf(S) ≠ eof do
    X ← topOf(S)
    if X = a then
      pop (S); a ← nextInputSymbol ()
    else if X is a terminal then
      Report an error;
    else if M[X, a] = X → Y₁ ... Yₙ then
      print (X → Y₁ ... Yₙ);
      pop (S);
      for i ← n to 1 do push (S, Yᵢ) ;
    else
      Report an error;
  end
end

stack: input:

output:

<table>
<thead>
<tr>
<th></th>
<th>id</th>
<th>+</th>
<th>*</th>
<th>( )</th>
<th>eof</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E’</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T’</td>
<td>6</td>
<td>(5)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(1) E → T E’
(2) E’ → + T E’
(3) | ε
(4) T → F T’
(5) T’ → * F T’
(6) | ε
(7) F → ( E )
(8) | id
Example of Nonrecursive Predictive Parsing

begin
a ← inputSymbol();
push (S, eof);
push (S, s0);
while topOf(S) ≠ eof do
X ← topOf(S);
if X = a then
    pop (S); a ← nextInputSymbol();
else if X is a terminal then
    Report an error;
else if M[X, a] = X → Y₁...Yₙ then
    print (X → Y₁...Yₙ);
    pop (S);
    for i ← n to 1 do push (S, Yᵢ);
else
    Report an error;
end
end

stack:

input:

output:

E → T E'
T → F T'
F → ( E )
id

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
Example of Nonrecursive Predictive Parsing

begin
  \( a \leftarrow \text{inputSymbol}() \);
  \( \text{push} (S, \text{eof}) \);
  \( \text{push} (S, s_0) \);
  \textbf{while} \( \text{top0f}(S) \neq \text{eof} \) \textbf{do}
    \( X \leftarrow \text{top0f}(S) \);
    \textbf{if} \( X = a \) \textbf{then}
      \( \text{pop} (S) ; a \leftarrow \text{nextInputSymbol}() \);
    \textbf{else if} \( X \) \textbf{is a terminal} \textbf{then}
      \( \text{Report an error} ; \)
    \textbf{else if} \( M[X, a] = X \rightarrow Y_1 \ldots Y_n \) \textbf{then}
      \( \text{print} (X \rightarrow Y_1 \ldots Y_n) ; \)
      \( \text{pop} (S) ; \)
      \( \text{for} \ i \leftarrow n \ \textbf{to} \ 1 \ \textbf{do} \ \text{push} (S, Y_i) ; \)
    \textbf{else}
      \( \text{Report an error} ; \)
  end
end

\[
\begin{array}{|c|c|c|c|c|}
\hline
\text{Symbol} & (1) & + & * & ( ) & \text{eof} \\
\hline
E & (1) & (1) & \hline
E' & (2) & (3) & (3) & \hline
T & (4) & (4) & \hline
T' & (6) & (5) & (6) & (6) & \hline
F & (8) & (7) & \hline
\end{array}
\]

\[
\begin{align*}
E & \rightarrow T \ E' \\
E' & \rightarrow + T \ E' \\
T & \rightarrow F \ T' \\
T' & \rightarrow * F \ T' \\
F & \rightarrow ( E ) \\
\end{align*}
\]

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
Example of Nonrecursive Predictive Parsing

```
begin
  a ← inputSymbol();
push (S, eof);
push (S, s₀);
while top0f(S) ≠ eof do
  X ← top0f (S);
  if X = a then
    pop (S); a ← nextInputSymbol ();
  else if X is a terminal then
    Report an error;
  else if M[X, a] = X → Y₁ ... Yₙ then
    print (X → Y₁ ... Yₙ);
pop (S);
    for i ← n to 1 do push (S, Yᵢ);
  else
    Report an error;
end
end
```

Input: `id + id * id eof`

Output:

```
E → T E'
T → F T'
F → id
```

Table:

<table>
<thead>
<tr>
<th></th>
<th>id</th>
<th>+</th>
<th>*</th>
<th>( )</th>
<th>eof</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>(1)</td>
<td></td>
<td></td>
<td>(1)</td>
<td></td>
</tr>
<tr>
<td>E'</td>
<td>(2)</td>
<td></td>
<td></td>
<td>(3)</td>
<td>(3)</td>
</tr>
<tr>
<td>T</td>
<td>(4)</td>
<td></td>
<td></td>
<td>(4)</td>
<td></td>
</tr>
<tr>
<td>T'</td>
<td>(6)</td>
<td>(5)</td>
<td></td>
<td>(6)</td>
<td>(6)</td>
</tr>
<tr>
<td>F</td>
<td>(8)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
Example of Nonrecursive Predictive Parsing

begin
  a ← inputSymbol();
  push (S, eof);
  push (S, s0);
  while topOf(S) ≠ eof do
    X ← topOf(S);
    if X = a then
      | pop (S); a ← nextInputSymbol();
    else if X is a terminal then
      | Report an error;
    else if M[X, a] = X → Y₁...Yₙ then
      | print (X → Y₁...Yₙ);
      | pop (S);
      | for i ← n to 1 do push (S, Yᵢ);
    else
      | Report an error;
  end
end

stack: input:

output:

<table>
<thead>
<tr>
<th></th>
<th>id</th>
<th>+</th>
<th>*</th>
<th>(</th>
<th>)</th>
<th>eof</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>(1)</td>
<td></td>
<td></td>
<td>(1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E'</td>
<td>(2)</td>
<td></td>
<td></td>
<td>(3)</td>
<td>(3)</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>(4)</td>
<td></td>
<td></td>
<td>(4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T'</td>
<td>(6)</td>
<td>(5)</td>
<td></td>
<td>(6)</td>
<td>(6)</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>(8)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(7)</td>
</tr>
</tbody>
</table>

(1) E → T E'
(2) E' → + T E'
(3)    | ε
(4) T → F T'
(5) T' → * F T'
(6)    | ε
(7) F → ( E )
(8)    | id
Example of Nonrecursive Predictive Parsing

begin
  a ← inputSymbol();
  push (S, eof);
  push (S, s0);
  while topOf(S) ≠ eof do
    X ← topOf(S);
    if X = a then
      pop (S); a ← nextInputSymbol();
    else if X is a terminal then
      Report an error;
    else if M[X, a] = X → Y1 ... Yn then
      print (X → Y1 ... Yn);
      pop (S);
      for i ← n to 1 do push (S, Yi);
    else
      Report an error;
  end
end

stack: input: output:

id + id * id eof

E    T E'
E'    + T E'
T    F T'
T'    * F T'
F    id
eof

E → T E'
E' → + T E'
T → F T'
T' → * F T'
F → id

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
Example of Nonrecursive Predictive Parsing

begin
a ← inputSymbol();
push (S, eof);
push (S, s0);
while top0f(S) ≠ eof do
    X ← top0f (S);
    if X = a then
        pop (S); a ← nextInputSymbol();
    else if X is a terminal then
        Report an error;
    else if M[X, a] = X → Y₁ Y₂ Yₙ then
        print (X → Y₁ Y₂ Yₙ);
        pop (S);
        for i ← n to 1 do push (S, Yᵢ);
    else
        Report an error;
end
end

<table>
<thead>
<tr>
<th></th>
<th>id</th>
<th>+</th>
<th>*</th>
<th>(</th>
<th>)</th>
<th>eof</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>(1)</td>
<td></td>
<td></td>
<td>(</td>
<td></td>
<td>)</td>
</tr>
<tr>
<td>E'</td>
<td>(2)</td>
<td></td>
<td></td>
<td>(</td>
<td></td>
<td>)</td>
</tr>
<tr>
<td>T</td>
<td>(4)</td>
<td></td>
<td></td>
<td>(</td>
<td></td>
<td>)</td>
</tr>
<tr>
<td>T'</td>
<td>(6)</td>
<td></td>
<td></td>
<td>(</td>
<td></td>
<td>)</td>
</tr>
<tr>
<td>F</td>
<td>(8)</td>
<td></td>
<td></td>
<td>(</td>
<td></td>
<td>)</td>
</tr>
</tbody>
</table>

stack:

input:

output:
Example of Nonrecursive Predictive Parsing

begin
  a ← inputSymbol();
  push (S, eof);
  push (S, s0);
  while top0f(S) ≠ eof do
    X ← top0f (S);
    if X = a then
      pop (S); a ← nextInputSymbol();
    else if X is a terminal then
      Report an error;
    else if M[X, a] = X → Y₁ ... Yₙ then
      print (X → Y₁ ... Yₙ);
      pop (S);
      for i ← n to 1 do push (S, Yᵢ);
    else
      Report an error;
  end
end

Example of Nonrecursive Predictive Parsing

<table>
<thead>
<tr>
<th></th>
<th>id</th>
<th>+</th>
<th>*</th>
<th>(</th>
<th>)</th>
<th>eof</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>(1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E'</td>
<td>(2)</td>
<td></td>
<td></td>
<td>(1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>(4)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T'</td>
<td>(6)</td>
<td>(5)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>(8)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>(1)</th>
<th>E → T E'</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2)</td>
<td>E' → + T E'</td>
<td></td>
</tr>
<tr>
<td>(3)</td>
<td></td>
<td>ε</td>
</tr>
<tr>
<td>(4)</td>
<td>T → F T'</td>
<td></td>
</tr>
<tr>
<td>(5)</td>
<td>T' → * F T'</td>
<td></td>
</tr>
<tr>
<td>(6)</td>
<td></td>
<td>ε</td>
</tr>
<tr>
<td>(7)</td>
<td>F → ( E )</td>
<td></td>
</tr>
<tr>
<td>(8)</td>
<td></td>
<td>id</td>
</tr>
</tbody>
</table>

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
Example of Nonrecursive Predictive Parsing

begin
  a ← inputSymbol ();
push (S, eof);
push (S, s0);
while top0f(S) ≠ eof do
  X ← top0f (S);
  if X = a then
    pop (S); a ← nextInputSymbol ();
  else if X is a terminal then
    Report an error;
  else if M[X, a] = X → Y1 ... Yn then
    print (X → Y1 ... Yn);
    pop (S);
    for i ← n to 1 do push (S, Yi);
  else
    Report an error;
end
end

---

<table>
<thead>
<tr>
<th></th>
<th>id</th>
<th>+</th>
<th>*</th>
<th>( )</th>
<th>eof</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>(1)</td>
<td></td>
<td></td>
<td></td>
<td>(1)</td>
</tr>
<tr>
<td>E'</td>
<td>(2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>(4)</td>
<td></td>
<td></td>
<td></td>
<td>(3)</td>
</tr>
<tr>
<td>T'</td>
<td>(6)</td>
<td>(5)</td>
<td></td>
<td>(6)</td>
<td>(6)</td>
</tr>
<tr>
<td>F</td>
<td>(8)</td>
<td></td>
<td></td>
<td>(7)</td>
<td></td>
</tr>
</tbody>
</table>

---

input: id + id * id eof

output: E → T E'
        T → F T'
        F → id
        T' → ε

---

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
Example of Nonrecursive Predictive Parsing

begin
begin

\begin{align*}
&\text{a ← inputSymbol();} \\
&\text{push (S, eof);} \\
&\text{push (S, s0);} \\
&\text{while top0f(S) \neq eof do} \\
&\quad X \leftarrow \text{top0f (S);} \\
&\quad \text{if } X = a \text{ then} \\
&\quad \quad \text{pop (S); a ← nextInputSymbol();} \\
&\quad \text{else if } X \text{ is a terminal then} \\
&\quad \quad \text{Report an error;} \\
&\quad \text{else if } M[X, a] = X \rightarrow Y_1 \ldots Y_n \text{ then} \\
&\quad \quad \text{print (X \rightarrow Y_1 \ldots Y_n);} \\
&\quad \quad \text{pop (S);} \\
&\quad \quad \text{for } i \leftarrow n \text{ to } 1 \text{ do push (S, } Y_i); \\
&\quad \text{else} \\
&\quad \quad \text{Report an error;}
end end
end

\begin{array}{|c|c|c|c|}
\hline
& \text{id} & + & \ast \\
\hline
E & (1) & & (1) \\
E' & (2) & & (3) (3) \\
T & (4) & & (4) \\
T' & (6) (5) & & (6) (6) \\
F & (8) & & (7) \\
\hline\end{array}

\begin{align*}
E & \rightarrow T E' \\
E' & \rightarrow + T E' \\
T & \rightarrow F T' \\
T' & \rightarrow \ast F T' \\
F & \rightarrow ( E ) \\
\end{align*}

\begin{align*}
(1) & E \rightarrow T E' \\
(2) & E' \rightarrow + T E' \\
(3) & \rightarrow \epsilon \\
(4) & T \rightarrow F T' \\
(5) & T' \rightarrow \ast F T' \\
(6) & \rightarrow \epsilon \\
(7) & F \rightarrow ( E ) \\
(8) & \rightarrow \text{id} \\
\end{align*}
Example of Nonrecursive Predictive Parsing

begin
a ← inputSymbol();
push (S, eof);
push (S, s0);
while top0f(S) ≠ eof do
    X ← top0f (S);
    if X = a then
        pop (S); a ← nextInputSymbol();
    else if X is a terminal then
        Report an error;
    else if M[X, a] = X → Y₁ ... Yₙ then
        print (X → Y₁ ... Yₙ);
        pop (S);
        for i ← n to 1 do push (S, Yᵢ);
    else
        Report an error;
end
end

stack:
input:

output:

<table>
<thead>
<tr>
<th></th>
<th>id</th>
<th>+</th>
<th>*</th>
<th>(</th>
<th>)</th>
<th>eof</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>(1)</td>
<td></td>
<td></td>
<td>(1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E'</td>
<td>(2)</td>
<td></td>
<td></td>
<td>(3)</td>
<td>(3)</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>(4)</td>
<td></td>
<td></td>
<td>(4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T'</td>
<td>(6)</td>
<td>(5)</td>
<td></td>
<td>(6)</td>
<td>(6)</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>(8)</td>
<td></td>
<td></td>
<td></td>
<td>(7)</td>
<td></td>
</tr>
</tbody>
</table>
Example of Nonrecursive Predictive Parsing

\[
\begin{align*}
\begin{array}{llllll}
| & E & + & * & ( & ) & \text{eof} \\
\hline
E & (1) & & & & \\
E' & & & & (1) & \\
T & (4) & & (3) & (3) & \\
T' & & (5) & & (6) & \\
F & (8) & & (6) & (6) & \\
\end{array}
\end{align*}
\]
Example of Nonrecursive Predictive Parsing

```plaintext
begin
  a ← inputSymbol();
  push (S, eof);
  push (S, s0);
  while top0f(S) ≠ eof do
    X ← top0f (S);
    if X = a then
      pop (S); a ← nextInputSymbol();
    else if X is a terminal then
      Report an error;
    else if M[X, a] = X → Y1 ... Yn then
      print (X → Y1 ... Yn);
      pop (S);
      for i ← n to 1 do push (S, Yi);
    else
      Report an error;
  end
end
```

```
# Example

<table>
<thead>
<tr>
<th></th>
<th>id</th>
<th>+</th>
<th>*</th>
<th>( )</th>
<th>eof</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>(1)</td>
<td></td>
<td></td>
<td>(1)</td>
<td></td>
</tr>
<tr>
<td>E'</td>
<td>(2)</td>
<td></td>
<td></td>
<td>(3)</td>
<td>(3)</td>
</tr>
<tr>
<td>T</td>
<td>(4)</td>
<td></td>
<td></td>
<td>(4)</td>
<td></td>
</tr>
<tr>
<td>T'</td>
<td>(6)</td>
<td>(5)</td>
<td></td>
<td>(6)</td>
<td>(6)</td>
</tr>
<tr>
<td>F</td>
<td>(8)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- (1) E → T E'
- (2) E' → + T E'
- (3) | ε
- (4) T → F T'
- (5) T' → * F T'
- (6) | ε
- (7) F → ( E )
- (8) | id
```

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
Example of Nonrecursive Predictive Parsing

begin
a ← inputSymbol ();
push (S, eof);
push (S, s0);
while top0f(S) ≠ eof do
X ← top0f (S);
if X = a then
    | pop (S); a ← nextInputSymbol ();
else if X is a terminal then
    | Report an error;
else if M[X, a] = X → Y₁ ... Yₙ then
    | print (X → Y₁ ... Yₙ);
    | pop (S);
    | for i ← n to 1 do push (S, Yᵢ) ;
else
    | Report an error;
end
end

stack:

input:

output:

<table>
<thead>
<tr>
<th></th>
<th>id</th>
<th>+</th>
<th>*</th>
<th>(</th>
<th>)</th>
<th>eof</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>(1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E'</td>
<td>(2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>(4)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T'</td>
<td>(6)</td>
<td>(5)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>(8)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(1) $E \rightarrow TE'$
(2) $E' \rightarrow + TE'$
(3) $\epsilon$
(4) $T \rightarrow FT'$
(5) $T' \rightarrow * FT'$
(6) $\epsilon$
(7) $F \rightarrow (E)$
(8) $id$
Example of Nonrecursive Predictive Parsing

begin
a ← inputSymbol();
push (S, eof);
push (S, s0);
while top0f(S) ≠ eof do
    X ← top0f(S);
    if X = a then
        pop (S); a ← nextInputSymbol();
    else if X is a terminal then
        Report an error;
    else if M[X, a] = X → Y₁ ... Yₙ then
        print (X → Y₁ ... Yₙ);
        pop (S);
        for i ← n to 1 do push (S, Yᵢ);
    else
        Report an error;
end
end

stack: input:
output:

id + id * id eof

stack: input:
output:
eof

E    T E'
E' → + T E'
T → F T'
F → id
T' → ε
E' → * F T'
T' → ε
F → (E)
(1) E → T E'
(2) E' → + T E'
(3) ε
(4) T → F T'
(5) T' → * F T'
(6) ε
(7) F → (E)
(8) id

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
Outline

1 Introduction

2 Context-free grammar

3 Parsing with a grammar
   - Top-down parsing
     - Principles
     - Recursive-descent parsing
     - FIRST and FOLLOW
     - $LL(1)$ grammars
     - Nonrecursive predictive parsing
     - Error recovery in predictive parsing
   - Bottom-up parsing
   - $LR$ parsing

4 Generate a syntactic parser with Yacc or JavaCC

5 Conclusion

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
Panic-mode error recovery is based on the idea of skipping over symbols on the input until a token in a selected set of synchronizing tokens appears.

Its effectiveness depends on the choice of synchronizing set.

The sets should be chosen so that the parser recovers quickly from errors that are likely to occur in practice.

Some heuristics are as follows:

1. As a starting point, place all symbols in \( \text{FOLLOW}(A) \) into the synchronizing set for nonterminal \( A \). If we skip tokens until an element of \( \text{FOLLOW}(A) \) is seen and pop \( A \) from the stack, it is likely that parsing can continue.
It is not enough to use FOLLOW(A) as the synchronizing set for A. We can add to the set of a lower-level construct the symbols that begin higher-level constructs. For example, we might add keywords that begin statements to the synchronizing sets for the nonterminals generating expressions.

If we add symbols in FIRST(A) to the synchronizing set for nonterminal A, then it may be possible to resume parsing according to A if a symbol in FIRST(A) appears in the input.

If a nonterminal can generate the empty string, then the production deriving $\epsilon$ can be used as a default. Doing so may postpone some error detection, but cannot cause an error to be missed. This approach reduces the number of nonterminals that have to be considered during error recovery.
If a terminal on top of the stack cannot be matched, a simple idea is to pop the terminal, issue a message saying that the terminal was inserted, and continue parsing. In effect, this approach takes the synchronizing set of a token to consist of all other tokens.
Outline

1. Introduction

2. Context-free grammar

3. Parsing with a grammar
   - Top-down parsing
   - Bottom-up parsing
     - Reductions
     - Handle pruning
     - Shift-reduce parsing
     - Conflicts during shift-reduce parsing
   - LR parsing

4. Generate a syntactic parser with Yacc or JavaCC

5. Conclusion

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
A bottom-up parsing corresponds to the construction of a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root.

This section introduces a general style bottom-up parsing known as shift-reduce parsing, which is attached the class of the $LR$ grammars.

$LR$ parsers are too difficult to be written by hand. We prefer to use automatic parser generators in place.
Outline

1. Introduction

2. Context-free grammar

3. Parsing with a grammar
   - Top-down parsing
   - Bottom-up parsing
     - Reductions
     - Handle pruning
     - Shift-reduce parsing
     - Conflicts during shift-reduce parsing
   - LR parsing

4. Generate a syntactic parser with Yacc or JavaCC

5. Conclusion
The bottom-up parsing is the process of “reducing” a string $w$ to the start symbol of the grammar.

At each reduction step, a specific substring matching the body of a production is replaced by the nonterminal at the head of that production.

The key decisions during bottom-up parsing are about when to reduce and about what production to apply, as the parse proceeds.

By definition, a reduction is the reverse of a step in a derivation. The goal of the bottom-up parsing is therefore to construct a derivation in reverse.
Let the grammar:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$E$</td>
<td>$\rightarrow$</td>
<td>$T$ $E'$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$E'$</td>
<td>$\rightarrow$</td>
<td>$+$ $T$ $E'$</td>
<td>$\mid$</td>
<td>$\epsilon$</td>
</tr>
<tr>
<td>$T$</td>
<td>$\rightarrow$</td>
<td>$F$ $T'$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$T'$</td>
<td>$\rightarrow$</td>
<td>$*$ $F$ $T'$</td>
<td>$\mid$</td>
<td>$\epsilon$</td>
</tr>
<tr>
<td>$F$</td>
<td>$\rightarrow$</td>
<td>$(E)$</td>
<td>$\mid$</td>
<td>id</td>
</tr>
</tbody>
</table>

A possible sequence of reductions is:

$\text{id} \ast \text{id} \leftarrow F \ast \text{id} \leftarrow F \ast F \leftarrow F \ast F \epsilon \leftarrow F \ast F T' \leftarrow F T'$
Example of Reductions

Let the grammar:

\[
\begin{align*}
E & \rightarrow T E' \\
E' & \rightarrow + T E' 
\quad \mid \epsilon \\
T & \rightarrow F T' \\
T' & \rightarrow * F T' 
\quad \mid \epsilon \\
F & \rightarrow ( E ) 
\quad \mid \text{id}
\end{align*}
\]

A possible sequence of reductions is:

\[
\text{id} \ast \text{id} \leftarrow F \ast \text{id} \leftarrow F \ast F \leftarrow F \ast F \epsilon \leftarrow F \ast F T' \leftarrow F T' \\
\leftarrow T \leftarrow T \epsilon \leftarrow T E' \leftarrow E
\]
What is the best sequence of reductions to build the parse tree?

- One method is to use the shift-reduce parsing method.

- The shift-reduce method is based on the handle pruning.
Outline

1. Introduction

2. Context-free grammar

3. Parsing with a grammar
   - Top-down parsing
   - Bottom-up parsing
     - Reductions
     - Handle pruning
     - Shift-reduce parsing
     - Conflicts during shift-reduce parsing
   - LR parsing

4. Generate a syntactic parser with Yacc or JavaCC

5. Conclusion

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
Informally, a **handle** is a substring that matches the body of a production, and whose reduction represents one step along the reverse of a rightmost derivation.

If $S \Rightarrow^* \alpha A \omega \Rightarrow^* \alpha \beta \omega$, then production $A \rightarrow \beta$ in the position following $\alpha$ is a handle of $\alpha \beta \omega$.

![Diagram](image)

A handle of a right-sentential form $\gamma$ is a production $A \rightarrow \beta$ and a position of $\gamma$ where the string $\beta$ may be found, such that replacing $\beta$ at that position by $A$ produces the previous right-sentential form in a rightmost derivation of $\gamma$.

Note that $\omega$ must contain only terminal symbols.
Algorithm of Handle Pruning

Input: A string of terminals $\omega$. A grammar $G$.
Output: A sequence of reductions of $\omega$, or an error if no sequence was found.

Hypothesis: $w = \gamma_n$, where $\gamma_n$ is the $n^{th}$ right-sentential form of some, yet unknown, rightmost derivation: $S = \gamma_0 \Rightarrow_{rm} \gamma_1 \Rightarrow_{rm} \gamma_2 \Rightarrow_{*} \gamma_{n-1} \Rightarrow_{rm} \gamma_n = \omega$

begin
    $d \leftarrow []$; $f \leftarrow \omega$;
    while $f \neq S$ do
        if $\exists h \in f | f = \alpha h \beta$; $\beta$ contains only terminals then
            if $\exists p \in G | p = (A \rightarrow h)$ then
                $f \leftarrow \alpha p \beta$;
                $d \leftarrow [(\alpha h \beta)].d$;
            else
                Throw("Cannot find a production for reduction");
            end
        else
            Throw("Cannot find a handle");
        end
    end
    return $d$;
end

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
1 Introduction

2 Context-free grammar

3 Parsing with a grammar
   - Top-down parsing
   - Bottom-up parsing
     - Reductions
     - Handle pruning
     - Shift-reduce parsing
     - Conflicts during shift-reduce parsing
   - LR parsing

4 Generate a syntactic parser with Yacc or JavaCC

5 Conclusion
Shift-reduce parsing is a form of bottom-up parsing in which a stack holds grammar symbols and an input buffer holds the rest of the string to be parsed.

The handle always appears at the top of the stack just before it is identified as the handle.

The four operations available during shift-reduce parsing are:

- **Shift**: Shift the next input symbol onto the top of the stack.
- **Reduce**: The right end of the string to be reduced must be at the top of the stack. Locate the left end of the string within the stack and decide with what nonterminal to replace the string.
- **Accept**: Announce successful completion of parsing.
- **Error**: Discover a syntax error and call an error recovery routine.
Example of Shift-Reduce Parsing

\[
\begin{align*}
E & \rightarrow T E' \\
E' & \rightarrow + T E' \\
& \quad | \epsilon \\
T & \rightarrow F T' \\
T' & \rightarrow * F T' \\
& \quad | \epsilon \\
F & \rightarrow ( E ) \\
& \quad | \text{id}
\end{align*}
\]
Example of Shift-Reduce Parsing

Stack:

```
stack: input:
```

```
E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → ( E ) | id
```

Input:
```
actions:
eof
```

Current:
```
Shift or Reduce?
Cannot reduce because the stack is empty.
Then: Shift
```
Example of Shift-Reduce Parsing

Context-free grammar

\[

e ightarrow \mathit{T} \ e' \\
e' \rightarrow + \mathit{T} \ e' | \epsilon \\
T \rightarrow \mathit{F} \ T' \\
T' \rightarrow * \mathit{F} \ T' | \epsilon \\
F \rightarrow ( \mathit{E} ) | \mathit{id}
\]

actions:

stack: input:

\[\text{id} \ast \text{id} \text{eof}\]
Example of Shift-Reduce Parsing

Context-free grammar

Stack: input:

\[
\begin{align*}
E & \rightarrow TE' \\
E' & \rightarrow +TE' \\
& \quad | \epsilon \\
T & \rightarrow FT' \\
T' & \rightarrow *FT' \\
& \quad | \epsilon \\
F & \rightarrow (E) \\
& \quad | id
\end{align*}
\]

Can Reduce?
Reduce with \( F \rightarrow id \)

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
Example of Shift-Reduce Parsing

\[
\begin{align*}
E &\rightarrow TE' \\
E' &\rightarrow + TE' | \epsilon \\
T &\rightarrow FT' \\
T' &\rightarrow * FT' | \epsilon \\
F &\rightarrow (E) | id
\end{align*}
\]

stack:

input:

actions:

Can Reduce? Reduce with \( F \rightarrow id \)
Example of Shift-Reduce Parsing

```
E → T E'  
E' → + T E' | ε  
T → F T'  
T' → * F T' | ε  
F → ( E ) | id
```

**Stack:**
```
[ ]
```

**Input:**
```
id * id eof
```

**Current:**
```
id * id
```

**Actions:**
```
Cannot Reduce.  
Then Shift.
```

\© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
Example of Shift-Reduce Parsing

```
E  →  T E'
E' →  + T E' | ε
T  →  F T'
T' →  * F T' | ε
F  →  ( E ) | id
```

**stack:**

```

```

**input:**

```
id * id eof
```

**current**

```

```

**actions:**

```
Cannot Reduce. Then Shift.
```

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
Example of Shift-Reduce Parsing

Context-free grammar:

\[

table: \begin{array}{|c|c|}
\hline
E & \rightarrow & TE' \\
E' & \rightarrow & + TE' \\
& & \epsilon \\
T & \rightarrow & FT' \\
T' & \rightarrow & * FT' \\
& & \epsilon \\
F & \rightarrow & (E) \\
& & id \\
\hline
\end{array}
\]

Actions:

id * id eof

Can Reduce? Reduce with F \rightarrow id
Example of Shift-Reduce Parsing

Context-free grammar

\[
\begin{align*}
E & \rightarrow TE' \\
E' & \rightarrow +TE' \mid \epsilon \\
T & \rightarrow FT' \\
T' & \rightarrow *FT' \mid \epsilon \\
F & \rightarrow (E) \mid \text{id}
\end{align*}
\]

Stack: 

```
stack: input: 
actions: 
eof
```

Input: 

```
stack: input: 
actions: 
eof
```

Cannot Reduce. Push $\epsilon$ before firing an error.
Example of Shift-Reduce Parsing

Context-free grammar

\[
\begin{align*}
E & \rightarrow TE' \\
E' & \rightarrow + TE' | \epsilon \\
T & \rightarrow FT' \\
T' & \rightarrow * FT' | \epsilon \\
F & \rightarrow (E) | id
\end{align*}
\]

Input: id * id eof

Stack:

Current

Actions:

Can Reduce?
Reduce with $T' \rightarrow \epsilon$
Example of Shift-Reduce Parsing

stack: input: actions:

\[
\begin{align*}
E & \rightarrow TE' \\
E' & \rightarrow + TE' | \epsilon \\
T & \rightarrow FT' \\
T' & \rightarrow * FT' | \epsilon \\
F & \rightarrow (E) | id
\end{align*}
\]

can Reduce? Reduce with \( T' \rightarrow * FT' \)

\[
\text{stack: } \text{input: } \text{actions:}
\]

\[
\text{id} \quad * \quad \text{id} \quad \text{eof}
\]

\[
\text{Can Reduce? Reduce with } T' \rightarrow * FT'
\]

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
Example of Shift-ReduceParsing

Context-free grammar

$E \rightarrow T E'$
$E' \rightarrow + T E'$
$E' \rightarrow \epsilon$
$T \rightarrow F T'$
$T' \rightarrow * F T'$
$T' \rightarrow \epsilon$
$F \rightarrow ( E )$
$F \rightarrow \text{id}$

stack: stack:

input: input:

actions: actions:

Can Reduce?
Reduce with $T \rightarrow F T'$

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
Example of Shift-Reduce Parsing

### Grammar

- \( E \rightarrow T \ E' \)
- \( E' \rightarrow + \ T \ E' \) | \( \epsilon \)
- \( T \rightarrow F \ T' \)
- \( T' \rightarrow * \ F \ T' \) | \( \epsilon \)
- \( F \rightarrow ( \ E \ ) \) | \( id \)

### Stack and Input

Stack: 
```
  input: id * id eof
```

Current: 
```
  actions: Cannot Reduce.
            Push \( \epsilon \) before firing an error.
```

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
Example of Shift-Reduce Parsing

**Grammar**

- \( E \rightarrow T E' \)
- \( E' \rightarrow + T E' \) | \( \epsilon \)
- \( T \rightarrow F T' \)
- \( T' \rightarrow * F T' \) | \( \epsilon \)
- \( F \rightarrow ( E ) \) | id

**Stack**

- input: id * id eof

**Actions**

- Can Reduce?
- Reduce with \( E' \rightarrow \epsilon \)

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
Example of Shift-Reduce Parsing

\[
\begin{align*}
E & \rightarrow TE' \\
E' & \rightarrow +TE' | \epsilon \\
T & \rightarrow FT' \\
T' & \rightarrow *FT' | \epsilon \\
F & \rightarrow (E) | \text{id} \\
\end{align*}
\]

Stack:

<table>
<thead>
<tr>
<th>E</th>
<th>T</th>
<th>E'</th>
</tr>
</thead>
</table>

Input:

- id * id eof

Actions:

- Can Reduce?
- Reduce with \( E \rightarrow TE' \).
- Then Accept.

Can Reduce?
Reduce with \( E \rightarrow TE' \).
Then Accept.
There are context-free grammars for which shift-reduce parsing cannot be used.

Every shift-reduce parser for such a grammar can reach a configuration in which the parser, knowing the entire stack and also the next \( k \) input symbols,

- cannot decide whether to shift or to reduce (shift/reduce conflict), or
- cannot decide which of several reductions to make (reduce/reduce conflict).

The grammars that cause these conflicts are generally non-LR grammars: they are not in the \( LR(k) \) class of grammars.
Consider the grammar:

\[
\text{statement} \rightarrow \text{if expression then statement} \\
\mid \text{if expression then statement else statement} \\
\mid \text{other}
\]

Consider the stack: `eof . . . if expression then statement`

Consider the input: `else . . . eof`

We cannot tell whether `if expression then statement` is the handle, no matter what appears below it on the stack. There is a shift/reduce conflict.

Depending on what follows the `else` on the input, it might be correct to reduce if-then to `statement`, or it might be correct to shift `else` and then to look for another `statement` to complete the if-then-else.
Consider the grammar with array indexes between parenthesis:

<table>
<thead>
<tr>
<th>Grammar</th>
<th>Production</th>
</tr>
</thead>
<tbody>
<tr>
<td>statement</td>
<td>→ id ( parameters )</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>parameters</td>
<td>→ parameters , id</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>expression</td>
<td>→ id ( expressions )</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>expressions</td>
<td>→ expressions , expression</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Consider the stack: `eof ... id ( id )`

Consider the input: `, id ) ... eof`

It is evident that the `id` on top of the stack should be reduced, but by which production?

1. `parameters → id` if `p` is a procedure, or
2. `expressions → id` if `p` is an array.
Introduction

Context-free grammar

Parsing with a grammar

Top-down parsing

Bottom-up parsing

LR parsing

Principles

LR(0) automaton

LR parsing algorithm

Building SLR-parsing table

LALR parsing

Generate a syntactic parser with Yacc or JavaCC

Conclusion

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
This section introduces a simple $LR$ (or $SLR$) parsing based on the concepts previously presented.

The most prevalent type of bottom-up parser is based on $LR(k)$ parsing:
- **L**eft-to-right input scanning,
- **R**ightmost derivation in reverse,
- $k$ input symbols are used for lookahead to make parsing action decisions.

In this chapter, only the cases of $k = 0$ or $k = 1$ are considered.
LR Parsers are table-driven, much like the nonrecursive LL parsers.

LR parser is attractive for:

1. LR parsers can be constructed to recognize virtually all programming language constructs for which context-free grammars can be written. Non-LR context-free grammars exist, but these can generally be avoided for typical programming-language constructs.

2. The LR-parsing method is the most general nonbacktracking shift-reduce parsing method. It can be implemented as efficiently as other, more primitive shift-reduce methods.

3. An LR parser can detect a syntactic error as soon as it is possible to do so on a left-to-right scan of the input.
The class of grammars that can be parsed using LR methods is a proper subset of the class of grammars that can be parsed with predictive or LL methods. For a grammar to be LR\((k)\), we must be able to recognize the occurrence of the right side of a production in a right-sentential form, with \(k\) input symbols of lookahead. This requirement is far less stringent than that for LL\((k)\) grammars where we must be able to recognize the use of a production seeing only the first \(k\) symbols of what its right side derives. Thus, it should not be surprising that LR grammars can describe more languages than LL grammars.
1 Introduction

2 Context-free grammar

3 Parsing with a grammar
   - Top-down parsing
   - Bottom-up parsing
   - LR parsing
     - Principles
     - LR(0) automaton
     - LR parsing algorithm
     - Building SLR-parsing table
     - LALR parsing

4 Generate a syntactic parser with Yacc or JavaCC

5 Conclusion
The LR(0) automaton helps with shift-reduce decisions.

Suppose that the string $\gamma$ of grammar symbols takes the $LR(0)$ automaton from the start state $I_0$ to some state $I_j$.

Then, shift on the next input symbol $a$ if state $I_j$ has a transition on $a$.

Otherwise, we choose to reduce; the items in state $I_j$ will tell us which production to use.
An LR parser makes shift-reduce decisions by maintaining states to keep track of where we are in a parse. States represent sets of “items.”

**Definition (Item)**

An LR(0) item (or item) of a grammar $G$ is a production of $G$ with a dot at some position of the body.

- The production $A \rightarrow XYZ$ yields the four items:
  
  \[
  \begin{align*}
  A & \rightarrow \bullet X Y Z \\
  A & \rightarrow X \bullet Y Z \\
  A & \rightarrow X Y \bullet Z \\
  A & \rightarrow X Y Z \bullet
  \end{align*}
  \]

- The production $A \rightarrow \epsilon$ generates only one item, $A \rightarrow \bullet$. 
Intuitively, an item indicates how much of a production we have seen at a given point in the parsing process.

For examples:

1. The item $A \rightarrow \bullet X Y Z$ indicates that we hope to see a string derivable from $XYZ$ next on the input.
2. The item $A \rightarrow X \bullet Y Z$ indicates that we have just seen on the input a string derivable from $X$ and that we hope next to see a string derivable from $YZ$.
3. The item $A \rightarrow X Y Z \bullet$ indicates that we have seen the body $XYZ$ and that it may be time to reduce $XYZ$ to $A$. 
Kernel items: The initial item, $S' \rightarrow S$, and all items whose dots are not at the left end.

Nonkernel items: All items with their dots at the left end, except for $S' \rightarrow S$. 
One collection of sets of $LR(0)$ items, called the canonical $LR(0)$ collection, provides the basis for constructing a deterministic finite automaton that is used to make parsing decisions.

Each state of the $LR(0)$ automaton represents a set of items in the canonical $LR(0)$ collection.

To construct the canonical $LR(0)$ collection for a grammar, we define:

1. an augmented grammar, and
2. Two functions, CLOSURE and GOTO.
Augmented Grammar

- If $G$ is a grammar with the start symbol $S$,

- Then $G'$, the augmented grammar of $G$, is $G$ with a new start symbol $S'$ and production $S' \rightarrow S$.

- The purpose of this new starting production is to indicate to the parser when it should stop parsing and announce acceptance of the input.

- That is, acceptance occurs when and only when the parser is about to reduce by $S' \rightarrow S$. 
If \( I \) is a set of items for a grammar \( G \), then \( \text{CLOSURE}(I) \) is the set of items constructed from \( I \) by the two rules:

1. Initially, add every item in \( I \) to \( \text{CLOSURE}(I) \)

2. If \( A \rightarrow \alpha \bullet B \beta \) is in \( \text{CLOSURE}(I) \) and \( B \rightarrow \gamma \) is a production, then add the item \( B \rightarrow \bullet \gamma \) to \( \text{CLOSURE}(I) \), if it not already there. Apply this rule until no more new items can be added to \( \text{CLOSURE}(I) \).
Example of CLOSURE

\[
\begin{array}{l}
E' & \rightarrow & E \\
E & \rightarrow & E + T \\
 & | & T \\
T & \rightarrow & T * F \\
 & | & F \\
F & \rightarrow & ( E ) \\
 & | & \text{id}
\end{array}
\]

\[
\begin{array}{c}
l_0 \\
E' \rightarrow \bullet E
\end{array}
\]

\[I = \{(E' \rightarrow \bullet E)\}\] and \(\text{CLOSURE}(I) = l_0\).
Consider $E$-productions because $E$ is on the right of the dot. Add $E \rightarrow \bullet E + T$ and $E \rightarrow \bullet T$ to $I_0$. 
Example of CLOSURE

Consider $E$-productions and $T$-productions because they are both on the right of the dot. Items for $E$-productions are already inside $I_0$, but not items for $T$-productions.
Definition (GOTO Function)

The closure of the set of all item $A \rightarrow \alpha \ X \bullet \ \beta$ such that $A \rightarrow \alpha \bullet \ X \ \beta$ is in $I$.

Where $I$ is a set of items, and $X$ is a grammar symbol.

- Intuitively, the GOTO function is used to define the transitions in the $LR(0)$ automaton for a grammar.
- The states of the automaton corresponds to sets of items, and $\text{GOTO}(I, X)$ specified the transition from the state $I$ under input $X$. 
Example of GOTO

If \( I \) is the set of two items \( \{(E' \rightarrow E\cdot), (E \rightarrow E \cdot + T)\} \)

Then, \( \text{GOTO}(I, +) = \text{CLOSURE}( \{(E \rightarrow E + \cdot T)\} \) is

\[
\begin{align*}
E' & \rightarrow E \\
E & \rightarrow E + T \\
& \quad | T \\
T & \rightarrow T * F \\
& \quad | F \\
F & \rightarrow (E) \\
& \quad | \text{id}
\end{align*}
\]

\[
\begin{align*}
E & \rightarrow E + \cdot T \\
T & \rightarrow \cdot T * F \\
& \quad | \cdot F \\
F & \rightarrow \cdot (E) \\
T & \rightarrow \cdot \text{id}
\end{align*}
\]
The Simple $LR$ (or $SLR$) parsing constructs the $LR(0)$ automaton from the grammar.

The states of this automaton are the sets of the items from the canonical $LR(0)$ collection, and the transitions are given by the GOTO function.

In the following slide, there is an example of $LR(0)$ automaton.

- Kernel items are in the light-yellow part of the box.
- Nonkernel items are in the dark-yellow part of the box.
- Edge represents the transitions given by the function GOTO, where the label is the token name.
Example of $LR(0)$ Automaton
Algorithm to Compute the Canonical $LR(0)$ Collection

**Input**: An augmented grammar $G'$.

```plaintext
begin
    $C \leftarrow \text{CLOSURE}(\{(S' \rightarrow \bullet S)\});$
    repeat
        foreach set of items $I \in C$ do
            foreach grammar symbol in $X$ do
                if $\text{GOTO}(I, X)$ is not empty and not in $C$ then
                    $C \leftarrow C \cup \text{GOTO}(I, X);$  
                end
            end
        end
    until no new sets of items are added to $C$ on a round;
    return $C$;
end
```
Outline

1. Introduction

2. Context-free grammar

3. Parsing with a grammar
   - Top-down parsing
   - Bottom-up parsing
   - LR parsing
     - Principles
     - LR(0) automaton
     - LR parsing algorithm
     - Building SLR-parsing table
     - LALR parsing

4. Generate a syntactic parser with Yacc or JavaCC

5. Conclusion

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
A LR parser consists of an input, an output, a stack, a driver program, and a parsing table that has two parts: ACTION and GOTO.

- Only the parsing table change from one parser to another.
- The parsing program reads characters from an input buffer one at a time.
- It shifts a state; not a symbol. This is a major difference between a LR parser and a shift-reduce parser.
The stack holds a sequence of states, $s_0 \ s_1 \ldots \ s_m$, where $s_m$ is on top. In SLR method, the stack holds the states from the $LR(0)$ automaton; the canonical $LR$ and $LALR$ methods are similar.

All the transition that are entering in a state are labeled with the same symbol. So that, a state may be associated to one symbol and only one, except for the start state.
This function takes as arguments a state \( s_j \) and a terminal \( a \) (or \( \text{eof} \)).

The value of \( \text{ACTION}[i, a] \) can have one of the four forms:

1. Shift \( j \), where \( s_j \) is a state. The action taken by the parser effectively shifts input \( a \) to the stack, but uses state \( s_j \) to represent \( a \).
2. Reduce \( A \rightarrow \beta \). The action of the parser effectively reduces \( \beta \) on the top of the stack to head \( A \).
3. Accept. The parser accepts the input and terminates.
4. Error. The parser discovers an error and takes some corrective action.
The GOTO function, defined on sets of items, is extended to states.

If $\text{GOTO}[l_i, A] = l_j$, then GOTO also maps a state $l_i$ and a nonterminal $A$ to state $l_j$. 
Algorithm for \( LR \)-Parsing

**Input**: An input string \( w \) and an \( LR \)-parsing table with functions \( ACTION \) and \( GOTO \) for a grammar \( G \).

**Output**: If \( w \) is in \( L(G) \), the reduction steps of a bottom-up parse for \( w \); otherwise, an error indication.

\[
\begin{align*}
\text{begin} & \quad S \leftarrow [s_0]; \quad a \leftarrow \text{inputSymbol}(); \quad \text{stopParser} \leftarrow \text{false}; \\
\text{while } \neg \text{stopParser} \text{ do} & \\
& \quad s \leftarrow \text{topOf} \( S \); \\
& \quad \text{if } ACTION[s, a] = \text{Shift} \( t \) \text{ then} \\
& \text{\hspace{1cm}} \quad \text{push} \( S, t \); \quad a \leftarrow \text{inputSymbol}(); \\
& \quad \text{if } ACTION[s, a] = \text{Reduce} \( A \rightarrow \beta \) \text{ then} \\
& \text{\hspace{1cm}} \quad \text{pop} \( S, \beta \); \quad t \leftarrow \text{topOf} \( S \); \quad \text{push} \( S, \text{GOTO}(t, A) \); \\
& \text{\hspace{1cm}} \quad \text{print} \( A \rightarrow \beta \); \\
& \quad \text{if } ACTION[s, a] = \text{Accept} \text{ then} \\
& \text{\hspace{1cm}} \quad \text{stopParser} \leftarrow \text{true}; \\
& \quad \text{else} \\
& \text{\hspace{1cm}} \quad \text{Throw "No production found"}; \\
\text{end} & \\
\text{end} &
\end{align*}
\]
Outline

1. Introduction

2. Context-free grammar

3. Parsing with a grammar
   - Top-down parsing
   - Bottom-up parsing
   - LR parsing
     - Principles
     - LR(0) automaton
     - LR parsing algorithm
     - Building SLR-parsing table
     - LALR parsing

4. Generate a syntactic parser with Yacc or JavaCC

5. Conclusion

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
The **SLR** method refers to the parsing table, the SLR table.

The **SLR** method begins with **LR(0)** items and **LR(0)** automata:

1. Given a grammar $G$, we augment $G$ to produce $G'$, with a new start symbol $S'$.
2. From $G'$, we construct $C$, the canonical collection of sets of items for $G'$ together with the GOTO function.
3. The ACTION and GOTO entries in the parsing table are then constructed using the algorithm in the following slides.
4. This algorithm requires us to know FOLLOW($A$) for each nonterminal $A$ of the grammar.
Algorithm for Building the \textit{SLR}-Parsing Table (#1)

- **INPUT:** An augmented grammar \( G' \).
- **OUTPUT:** The \textit{SLR}-parsing table functions \textit{ACTION} and \textit{GOTO} for \( G' \).
- **METHOD:**

1. Construct \( C = \{ I_0, I_1, \ldots, I_n \} \), the collection of sets of \( LR(0) \) items for \( G' \).
2. State \( s_i \) is constructed from \( I_i \). The parsing actions for state \( s_i \) are determined as follows:
   - If \( A \rightarrow \alpha \bullet \alpha \beta \) is in \( I_i \) and \( \text{GOTO}(I_i, a) = I_j \), then set \( \text{ACTION}[i, a] \) to “Shift \( j \).” Here \( a \) must be a terminal.
   - If \( A \rightarrow \alpha \bullet \alpha \) is in \( I_i \), then set \( \text{ACTION}[i, a] \) to “Reduce \( A \rightarrow \alpha \)” for all \( a \) in \( \text{FOLLOW}(A) \); here \( A \) may not be \( S' \).
Algorithm for Building the SLR-Parsing Table (#2)

- If $S' \rightarrow S \bullet$ is in $I_i$, then set ACTION[$i, a$] to “Accept.”

If any conflicting actions result from the above rules, we say that grammar is not SLR(1). The algorithm fails to produce a parser in this case.

3 The goto transitions from state $s_j$ are constructed for all nonterminals $A$ using the rule: If GOTO($l_i, A$) = $l_j$, then GOTO($s_i, A$) = $s_j$.

4 All entries not defined by rules (2) and (3) are made “error.”

5 The initial state of the parser is the one constructed from the set of items containing $S' \rightarrow S \bullet$. 
1 Introduction

2 Context-free grammar

3 Parsing with a grammar
   - Top-down parsing
   - Bottom-up parsing
   - LR parsing
     - Principles
     - LR(0) automaton
     - LR parsing algorithm
     - Building SLR-parsing table
   - LALR parsing

4 Generate a syntactic parser with Yacc or JavaCC

5 Conclusion
The $LALR$ parsing, or LookAhead $LR$ parsing, is often used in practice.

This section briefly describes the $LALR$ parsing without going deep in details. For details the reader should consult the list of books of this chapter.

The tables obtained by $LALR$ methods are significantly smaller than the tables obtained by canonical $LR$ methods.

The same is true for $SLR$ method, but $SLR$ methods cannot handle conveniently all the programming language constructs.

$LALR$ parsers offer many of the advantages of $SLR$ and canonical-$LR$ parsers.
They combine the states that have the same kernels (sets of items, ignoring the associated lookahead sets).

Thus, the number of states is the same as that of the SLR parser, but some parsing-action conflicts present in the SLR parser may be removed in the LALR parser.

LALR parsers have become the method of choice in practice.
Outline

1. Introduction
2. Context-free grammar
3. Parsing with a grammar
4. Generate a syntactic parser with Yacc or JavaCC
   - Overview
   - Yacc/Bison
   - JavaCC
5. Conclusion
Parser generators such as Yacc and its more recent implementation Bison, are generally LALR parser generators.

They permit to facilitate the creation of the front-end of a compiler by generating the source code from a grammar and a lexical analyzer specification.

This section describes two parsers generators:

- Yacc, or Bison, that generates C and C++ parsers.
- JavaCC, that generates Java parsers.
1 Introduction

2 Context-free grammar

3 Parsing with a grammar

4 Generate a syntactic parser with Yacc or JavaCC
   - Overview
   - Yacc/Bison
     - Using Yacc/Bison
     - Ambiguous grammar
     - Connecting to Lex
     - Error recovery
   - JavaCC

5 Conclusion
Process of Yacc

Yacc Program: file.y

C Program: file.tab.c

Yacc Compiler

C Compiler

C Program: file.tab.c

C Program: main.c

Executable: a.out

Parser: a.out

Input Stream: a.out

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
A Yacc program has the following form:

- Declarations
- Translation rules
- Auxiliary functions

Declarations

- C ordinary declarations, between \%{ and \%}.
- Declarations of tokens with the command \%token.
A Yacc program has the following form:

- Declarations
- Translation rules
- Auxiliary functions

Translation rules

Each rule consists of a grammar production and the associated action (note the final semicolon).

\[
\text{\textless head\textgreater} : \begin{cases} 
\text{\textless body}_1\textgreater \{ \text{\textless action}_1\textgreater \} \\
| \begin{cases} 
\text{\textless body}_2\textgreater \{ \text{\textless action}_2\textgreater \} \\
\begin{cases} 
\text{\textless body}_n\textgreater \{ \text{\textless action}_n\textgreater \} \\
\end{cases} \\
\end{cases}
\end{cases}
\]
A Yacc program has the following form:

```
<table>
<thead>
<tr>
<th>Declarations</th>
</tr>
</thead>
<tbody>
<tr>
<td>% %</td>
</tr>
<tr>
<td>Translation rules</td>
</tr>
<tr>
<td>% %</td>
</tr>
<tr>
<td>Auxiliary functions</td>
</tr>
</tbody>
</table>
```

**Auxiliary functions**

- Auxiliary functions are the section where additional C routines should be put.
- Note that you must provide the function `yylex()`, which is invoking the lexical analyzer (explained later).
Example of a Yacc Program

```c
%{
#include <ctype.h>
%
}
%token DIGIT

line : expr newline { printf("%d\n", $1); }
expr  : expr '+' term   { $$ = $1 + $3; }
     | term         ;
term  : term '*' factor  { $$ = $1 * $3; }
     | factor       ;
factor : '(' expr ')'   { $$ = $2; }
     | DIGIT        ;
%

int yylex() {
    int c;
    c = getchar();
    if (isdigit(c)) {
        yylval = c - '0'; /* convert char to int */
        return DIGIT;
    }
    return c;
}
```
Outline

1. Introduction
2. Context-free grammar
3. Parsing with a grammar
4. Generate a syntactic parser with Yacc or JavaCC
   - Overview
   - Yacc/Bison
     - Using Yacc/Bison
     - Ambiguous grammar
     - Connecting to Lex
     - Error recovery
   - JavaCC
5. Conclusion
Yacc provides a set of declarations that may be used to remove grammar ambiguity.

Associativity and Precedence:

- **Left associativity:** %left <op1> <op2>…
- **Right associativity:** %right <op3> <op4>…
- **No associativity:** %nonassoc <op5> <op6>…
- The tokens are given precedences in the order in which they appear in the declaration part, lower first.
- Tokens in the same declaration have the same precedence.
Outline

1. Introduction

2. Context-free grammar

3. Parsing with a grammar

4. Generate a syntactic parser with Yacc or JavaCC
   - Overview
   - Yacc/Bison
     - Using Yacc/Bison
     - Ambiguous grammar
     - Connecting to Lex
   - Error recovery
   - JavaCC

5. Conclusion

© 2012–2014 Stéphane Galland  
stephane.galland@utbm.fr – http://www.multiagent.fr
Lex was designed to produce lexical analyzers that could be used with Yacc.

The Lex library provides a driver program named yylex().

To use Lex in Yacc, you must remove any definition of yylex() in the Yacc specification; and replace this definition by:

```c
#include "lex.yy.c"
```

All the tokens defined in the Yacc declarations are directly available in the Lex program.
1 Introduction

2 Context-free grammar

3 Parsing with a grammar

4 Generate a syntactic parser with Yacc or JavaCC
   - Overview
   - Yacc/Bison
     - Using Yacc/Bison
     - Ambiguous grammar
     - Connecting to Lex
     - Error recovery
   - JavaCC

5 Conclusion

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
In Yacc, error recovery uses a form of error productions.

First, you must decide what “major” nonterminals will have error recovery associated to them.

Typical choices are some subset of the nonterminals generating expressions, statements, blocks, and functions.
Error Recovery Functions

- `yyerror()` reports an error.
- `yyerrok()` resets the parser to its normal mode of operation.

Here, the error production causes the program to suspend normal parsing when a syntax error is found on an input line.

On encountering the error, the parser in the program starts popping symbols from its stack until it encounters a statement that as a shift action on the token error. Then the input is read until the new-line character is read. Then the parser reduces error `'\n'` to lines, and emits the diagnostic message “error message.”

```
line :   lines expr '\n' { printf("%g\n", $2); }  
      |   lines '\n'  
      |   /* empty or epsilon */  
      |   error '\n'   { yyerror("error_message"); yyerrok(); }  
```
Outline

1. Introduction
2. Context-free grammar
3. Parsing with a grammar
4. Generate a syntactic parser with Yacc or JavaCC
   - Overview
   - Yacc/Bison
   - JavaCC
     - Using to JavaCC
     - Error recovery
5. Conclusion

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
Process of JavaCC

JavaCC Program: file.jj

JavaCC Compiler

Java Program: file.java

Java Program: file.java

Java Compiler

Executable: parser.class

Java Program: main.java

Java Compiler

Executable: parser.class

Input Stream

parser.class

Parser

output
A JavaCC program has the following form:

```
JavaCC options
PARSER_BEGIN(<parserName>)
Java compilation unit
PARSER_END(<parserName>)
Translation rules
```

Parser Definition

- The name that follows “PARSER_BEGIN” and “PARSER_END” must be the same and this identifies the name of the generated parser.
A JavaCC program has the following form:

```
JavaCC options
PARSER_BEGIN(<parserName>)
Java compilation unit
PARSER_END(<parserName>)
Translation rules
```

**Options**

- JavaCC options permits to control the behavior of the parser.
A JavaCC program has the following form:

```
JavaCC options
PARSER_BEGIN(<parserName>)
Java compilation unit
PARSER_END(<parserName>)
Translation rules
```

**Translation rules**

The Java compilation unit is a Java code that must contain at least the declaration of the class of the parser:

```
...  
public class <parser_name> {  
...  
}  
...  
```
A JavaCC program has the following form:

<table>
<thead>
<tr>
<th>JavaCC options</th>
</tr>
</thead>
<tbody>
<tr>
<td>PARSER_BEGIN(&lt;parserName&gt;)</td>
</tr>
<tr>
<td>Java compilation unit</td>
</tr>
<tr>
<td>PARSER_END(&lt;parserName&gt;)</td>
</tr>
<tr>
<td>Translation rules</td>
</tr>
</tbody>
</table>

**Predefined functions**

Two functions are automatically generated inside the parser class:

- **Token getNextToken()**: returns the next available token.
- **Token getToken(int index)**: returns the ith token ahead.
A JavaCC program has the following form:

```
JavaCC options
PARSER_BEGIN(<parserName>)
Java compilation unit
PARSER_END(<parserName>)
Translation rules
```

Translation rules

- Java code production (see error recovery for an example),
- Regular expression production,
- BNF production, or
- Token manager declarations (not treated in this lecture).
Define the Regular Expression Productions

Definition

[<state_list>] <kind> [IGNORE_CASE ]:
{ <regexp> | <regexp> | ... }

- `<state_list>` specifies the lexer states in which the rule is enabled (default is DEFAULT).
- `IGNORE_CASE` specifies, by its presence, that if the regular expression is case sensitive or case insensitive.
- The regular definitions are defined and used as follows, respectively (The ”#” before the id indicates that this definition exists solely for the purpose of defining other tokens):
  
  `< [# ]id : regexpr >
  `< id>`
**Types of Regular Expression Productions** \(<\text{kind}>\)

1. **TOKEN**: describes tokens in the grammar. The token manager creates a Token object for each match of such a regular expression and returns it to the parser.

2. **SPECIAL_TOKEN**: like tokens, except that they do not have significance during parsing, i.e., the BNF productions ignore them.

3. **SKIP**: simply skipped (ignored) by the token manager.

4. **MORE**: Sometimes it is useful to gradually build up a token to be passed on to the parser. Matches to this kind of regular expression are stored in a buffer until the next **TOKEN** or **SPECIAL_TOKEN** match.
Attributes of the Predefined Token Class

- **int** `kind`
  This is the index for this kind of token in the internal representation scheme of JavaCC. It may be replaced by a constant.

- **int** `beginLine`, `beginColumn`, `endLine`, `endColumn`
  The beginning and ending positions of the token as it appeared in the input stream.

- **String** `image`
  The image of the token as it appeared in the input stream.

- **Token** `next`
  A reference to the next regular (non-special) token from the input stream.
Object getValue()
An optional attribute value of the Token. Tokens which are not used as syntactic sugar will often contain meaningful values that will be used later on by the compiler or interpreter. This attribute value is often different from the image. Any subclass of Token that actually wants to return a non-null value can override this method as appropriate.

static final Token newToken(int ofKind)
static final Token newToken(int ofKind, String image)
Returns a new token object as its default behavior.
The name of the non-terminal is the name of the method, and the parameters and return value declared are the means to pass values up and down the parse tree.

Non-terminals on the right hand sides of productions are written as method calls, so the passing of values up and down the tree are done using exactly the same paradigm as method call and return.

The default access modifier for BNF productions is public.
The name of the non-terminal is the name of the method, and the parameters and return value declared are the means to pass values up and down the parse tree.

Non-terminals on the right hand sides of productions are written as method calls, so the passing of values up and down the tree are done using exactly the same paradigm as method call and return.

The default access modifier for BNF productions is public.
Definition

\(<\text{access}\_\text{modifier}>\) <\text{return}\_\text{type}> \\
<\text{identifier}> ( <\text{parameters}> ) : \\
<\text{java}\_\text{block}> \\
\{ <\text{expansion}\_\text{choices}> \}

- **Java block**: arbitrary Java declarations and code put at the beginning of the method generated for the Java non-terminal.
- **Expansion choices**: a sequence of expansion units. Each nonterminal is written as a function call. Semantic actions are Java blocks inside this part of the BNF production.
Example of JavaCC File (#1)

```java
PARSER_BEGIN(CalculatorParser)
    public class CalculatorParser {
    }
PARSER_END(CalculatorParser)

SKIP: {
    " "
    | " \t"
    | " \n"
    | " \r"
}

TOKEN:
{
    <DIGIT : [0–9]>
}
```
private void line() :
{
    int e;
}
{    e = expr(); { System.out.println(e); } }

private int expr() :
{
    int e, t;
}
{    e = expr() "+" t = term() { return e+t; }
|    t = term() { return v; }
}
Example of JavaCC File (#3)

```java
private int term() : {
    int t, f;
}  
| t = term() "*" f = factor()  { return t*f; }
| f = factor()  { return f; }  

private int factor() : {
    int e, d;
}  
| "(" e = expr () ")"  { return e; }
| d = <DIGIT>  { return Integer.parseInt(d.image); }
```
1 Introduction

2 Context-free grammar

3 Parsing with a grammar

4 Generate a syntactic parser with Yacc or JavaCC
   • Overview
   • Yacc/Bison
     • JavaCC
       • Using to JavaCC
       • Error recovery

5 Conclusion
Simply modify the file `ParseException.java` to do what you want it to do. Typically, you would modify the `getMessage` method to do your own customized error reporting.

All information regarding these methods can be obtained from the comments in the generated files `ParseException.java` and `TokenMgrError.java`.

There is a method in the generated parser called `generateParseException()`.

You can call this method anytime you wish to generate an object of type `ParseException`. This object will contain all the choices that the parser has attempted since the last successfully consumed token.
JavaCC offers two kinds of error recovery:

1. **Shallow recovery**: recovers if none of the current choices have succeeded in being selected.

2. **Deep recovery**: is when a choice is selected, but then an error happens sometime during the parsing of this choice.
When no token found, we want to skip until the next given symbol (semi-column).

```
TOKEN : { <SEMICOLON: ";" > }
private void stm() {
{}  
  ifStm()  
  | whileStm()
}

TOKEN : { <SEMICOLON: ";" > }
private void stm() {
{}  
  ifStm()  
  | whileStm()  
  | whileStm()  
  | error.skipTo(SEMICOLON)
}
```
- `error_skipto()` is a nonterminal that must be define prior to its first usage.

- To do so, we must use the following JAVACODE rule.

```java
void error_skipto(int kind) {
    ParseException e = generateParseException()
    System.err.println(e);
    Token t;
    do {
        t = getNextToken();
    } while (t.kind != kind);
}
```
When error occurred (even deeper in the parse tree), we want to recover.

```java
TOKEN : { <SEMICOLON: ";" > }
private void stm() : 
{} 
{ ifStm() 
  | whileStm() 
}

TOKEN : { <SEMICOLON: ";" > }
private void stm() : 
{} 
{ try 
  { ifStm() 
    | whileStm() 
    | whileStm() 
  } catch(ParseException e) { 
    error_skipto(e, SEMICOLON); 
  } 
}
Definition of the function `error_skipto()`

```
JAVACODE
void error_skipto(ParseException e, int kind) {
    System.out.println(e);
    Token t;
    do {
        t = getNextToken();
    } while (t.kind != kind);
}
```
1. Introduction

2. Context-free grammar

3. Parsing with a grammar

4. Generate a syntactic parser with Yacc or JavaCC

5. Conclusion
Key Concepts in the Chapter (#1)

- **Parsers**: A parser takes as input tokens from the lexical analyzer and treats the token names as terminal symbols of a context-free grammar. The parser then constructs a parse tree for its input sequence of tokens; the parse tree may be constructed figuratively or literally.

- **Context-Free Grammars**: A grammar specifies a set of terminal symbols (inputs), another set of nonterminals (symbols representing syntactic constructs), and a set of productions, each of which gives a way in which strings represented by one nonterminal can be constructed from terminal symbols and strings represented by certain other nonterminals. A production consists of a head (the nonterminal to be replaced) and a body (the replacing string of grammar symbols).
Key Concepts in the Chapter (#2)

- **Derivations**: The process of starting with the start-nonterminal of a grammar and successively replacing it by the body of one its productions is called derivation. If the leftmost (or rightmost) nonterminal is always replaced, then the derivation is called leftmost (resp. rightmost.)

- **Parse Trees**: A parse tree is a picture of a derivation, in which there is a node for each nonterminal that appears in the derivation. The children of a node are the symbols by which that nonterminal is replaced in the derivation. There is a one-to-one correspondence between parse trees, leftmost derivation, and rightmost derivations of the same terminal string.

- **Ambiguity**: A grammar for which some terminal string has two or more different parse tree is said to be ambiguous.
Top-Down and Bottom-Up Parsing: Parsers are generally distinguished by whether they work top-down or bottom-up. Top-down parsers include recursive-descent and LL parsers, while the most common forms of bottom-up parsers are LR parsers.

Design of Grammars: Grammars suitable for top-down parsing often are harder to design than those used by bottom-up parsers. It is necessary to eliminate left-recursion. We also must left-factor/group productions for the same nonterminal that have a common prefix in the body.

Recursive-Descent Parsers: These parsers use a procedure for each nonterminal.
**Key Concepts in the Chapter (#4)**

- **LL(1) Parsers:** A grammar such that it is possible to choose the correct production with which to expand a given nonterminal, looking only at the next input symbol, is called LL(1). These grammars allow us to construct a predictive parsing table that gives, for each nonterminal and each lookahead symbol, the correct choice of production.

- **Shift-Reduce Parsing:** Bottom-up parsers generally operate by choosing on the basis of the next input symbol and the contents of the stack, whether to shift the next input onto the stack, or to reduce some symbols at the top of the stack. A reduce takes a production body at the top of the stack and replaces it by the head of the production.
**Viable Prefixes:** In shift-reduce parsing, the stack contents are always a viable prefix, i.e. a prefix of some right-sentential form that ends no further right than the end of the handle of that right-sentential form. The handler is the substring that was introduced in the last step of the rightmost derivation of that sentential form.

**Valid Items:** An item is a production with a dot somewhere in the body. An item is valid for a viable prefix if the production of that item is used to generate the handler, and the viable prefix includes all those symbols to the left of the dot.

**LR Parsers:** Each of the several kinds of LR parsers operate by first constructing the sets of valid items (called LR states) for all possible viable prefixes, and keeping track of the state for each prefix on the stack. The set of valid items guide the shift-reduce parsing decision.
Simple LR Parsers: In an SLR parser, we perform a reduction implied by a valid item with a dot at the right end, provided the lookahead symbol can follow the head of that production in some sentential form.

Canonical-LR Parsers: This more complex form of LR parser uses items that are augmented by the set of lookahead symbols that can follow the use of the underlying production. A canonical-LR parser can avoid some of the parsing-action conflicts that are present in SLR parsers, but often has many more states than the SLR parser for the same grammar.
Bibliography of the Chapter (#1)

The syntax and semantics of the proposed international algebraic language of the zurich-ACM-GAMM conference.

Parsing algorithms with backtrack.
Information and Control, 23:1–34.

On the ambiguity problem of backus systems.
In J. ACM, volume 9, pages 477–479. ACM.

Three models for the description of language.

Bibliography on syntax error handling in language translation systems.
Available from the comp.compilers newsgroup.

Practical Translators for LR(k) Languages.
PhD thesis, MIT, Cambridge, MA.
Simple $LR(k)$ grammars.

An efficient context-free parsing algorithm.

On the ambiguity in phase-structure languages.
In *Comm. ACM*, volume 5, pages 526–534. ACM.

Report on the elliott ALGOLtranslator.

*Introduction to Automata Theory, Languages, and Computation.*
Addison-Wesley, Boston, MA.

Ingerman, P. (1967).
Panini-backus form suggested.
In *Comm. ACM*, volume 10, page 137. ACM.

Yacc - yet another compiler compiler.


Younger, D. (1967). Recognition and parsing of context-free languages in time $n^3$.

This chapter develops two major points:

1. The translation of languages guided by context-free grammars.
2. These translation techniques are applied to type checking and intermediate code generation.
A syntax-directed translation specifies the values of attributes, attached to the grammar symbols, by associating semantic rules with the grammar productions.

- A Syntax-directed translation scheme embeds program fragments (or semantic actions, between braces) inside the semantic rules.

\[
\begin{align*}
expr & \rightarrow expr + term \\
& \quad | \quad expr - term \\
& \quad | \quad term \\
\text{term} & \rightarrow 0 \\
& \quad | \quad 1 \\
& \quad | \quad 9 \\
\text{head.t} & = \begin{cases} 
\text{expr.t} \parallel \text{term.t} \parallel '+' \\
\text{expr.t} \parallel \text{term.t} \parallel '-' \\
\text{term.t} \\
\text{'}0' \\
\text{'}1' \\
\text{'9'} \\
\end{cases}
\end{align*}
\]
The most general approach to syntax-directed translation is:
- to construct a parse tree, and
- then to compute the values of the attributes at the nodes of the tree by visiting the nodes.

In many cases, translation can be done during parsing, without building an explicit tree.

Two classes of syntax-directed translation are presented:
- L-attributed translations (L=left)
- S-attributed translations (S=synthesized)
Static checking includes type checking, which ensures that operators are applied to compatible operands.

It also includes any syntactic checks that remain after parsing.
In the process of translating a program in a given source language into code for a given target machine, a compiler may construct a sequence of intermediate representations.

High-level representation is close to the source language e.g., syntax tree.

Low-level representation are close to the target machine e.g., three-address code.
1 Introduction

2 Translation scheme
   - Syntax-directed translation
   - Syntax-directed definition
   - Attributes of the productions
   - Evaluating a SDD with a parse tree
   - Dependency graph
   - S-attributed definition
   - L-attributed definition

3 Syntax tree and graph

4 Three-address code
Syntax-Directed Translation

Definition

Syntax-directed translation is done by attaching rules or program fragments to productions in a grammar.

- The rules or program fragments are named “semantic rules.”
- Each of these semantic rules permits to translate the source program into the target program.

Example: Translate infix arithmetic expressions into postfix arithmetic expressions.

Two concepts are related to syntax-directed translation: attributes and translation schemes.
1 Introduction

2 Translation scheme
   - Syntax-directed translation
   - Syntax-directed definition
     - Attributes of the productions
     - Evaluating a SDD with a parse tree
     - Dependency graph
     - S-attributed definition
     - L-attributed definition

3 Syntax tree and graph

4 Three-address code
Definition (Syntax-Directed Definition — SDD)

A context-free grammar together with attributes and rules.

- Attributes are associated with grammar symbols.
- Rules are associated with productions.

<table>
<thead>
<tr>
<th>Productions</th>
<th>Semantic Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>expr → expr + term</td>
<td>head.t = expr.t</td>
</tr>
<tr>
<td>expr - term</td>
<td>head.t = expr.t</td>
</tr>
<tr>
<td>term</td>
<td>head.t = term.t</td>
</tr>
<tr>
<td>0</td>
<td>head.t = '0'</td>
</tr>
<tr>
<td>1</td>
<td>head.t = '1'</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>9</td>
<td>head.t = '9'</td>
</tr>
</tbody>
</table>
# Two Notations for Syntax-Directed Definition

## Notation during the lectures:

<table>
<thead>
<tr>
<th>Productions</th>
<th>Semantic Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A \rightarrow B C$</td>
<td>First part of Rule 1</td>
</tr>
<tr>
<td>$D E$</td>
<td>Second part of Rule 1</td>
</tr>
<tr>
<td>$F$</td>
<td>Rule 2</td>
</tr>
<tr>
<td>$G \rightarrow H I J$</td>
<td>Rule 3</td>
</tr>
</tbody>
</table>

## Notation during the tutorial sessions and labworks:

```
A  →  B C { First part of Rule 1 } D E { Second part of Rule 1 }
|   F { Rule 2 }
G  →  H I J { Rule 3 }
```
1 Introduction

2 Translation scheme
   - Syntax-directed translation
   - Syntax-directed definition
   - Attributes of the productions
     - Evaluating a SDD with a parse tree
     - Dependency graph
     - S-attributed definition
     - L-attributed definition

3 Syntax tree and graph

4 Three-address code
An attribute is any quantity associated with a programming construct.

Examples: data types, number of instructions, line of the first occurrence of an identifier... 

Since we use grammar symbols (terminals and nonterminals) to represent programming constructs, we extend the notion of attributes from constructs to the symbols that represent them.
In this lecture, the attributes are written as one of:

- `<terminal>..<attribute name>`; or
- `<nonterminal>..<attribute name>.

The keyword head represents the nonterminal in the head of the rule.

If the same nonterminal is present many times in the same rule body, the attribute prefix is indexed by the position of the nonterminal in this body.

\[
\begin{align*}
expr & \rightarrow expr + expr \\
& \quad | expr - expr \\
& \quad | term \\
term & \rightarrow 0 \\
& \quad | 1 \\
& \quad | \ldots \\
& \quad | 9
\end{align*}
\]
Synthetized Attribute

Definition

A synthesized attribute for a nonterminal $A$ at a parse-tree node $N$ is defined by a semantic rule associated with the production at $N$.

- Note that the production must have $A$ as its head.
- A synthesized attribute at node $N$ is defined only in terms of attribute values at the children of $N$ and at $N$ itself.
Definition

An inherited attribute for a nonterminal $B$ at a parse-tree node $N$ is defined by a semantic rule associated with the production at the parent of $N$.

- Note that the production must have $B$ as a symbol in its body.

- An inherited attribute at node $N$ is defined only in terms of attribute values at $N$’s parent, $N$ itself, and $N$’s siblings.
1 Introduction

2 Translation scheme
   - Syntax-directed translation
   - Syntax-directed definition
   - Attributes of the productions
   - Evaluating a SDD with a parse tree
   - Dependency graph
   - S-attributed definition
   - L-attributed definition

3 Syntax tree and graph

4 Three-address code

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
To visualize the translation specified by an SDD, it helps to work with parse trees.

- A parse tree, showing the value(s) of its attribute(s) is called an annotated parse tree.

- Before we can evaluate an attribute at a node of a parse tree, we must evaluate all the attributes upon which its value depends.

- With synthesized attributes, we can evaluate attributes in any bottom-up order, such that of a postorder traversal of the parse tree.
Example of Evaluation

```
\[
\begin{align*}
T' \rightarrow & \quad F \; T' \\
T' \rightarrow & \quad * \; F \; T' \\
F \rightarrow & \quad \text{digit}
\end{align*}
\]
```

Input: 3 * 5

```
\[
\begin{align*}
T.\text{val} &= \text{?} \\
F.\text{val} &= \text{?} \\
T'.\text{lval} &= \text{?} \\
T'.\text{val} &= \text{?} \\
\text{digit}.\text{lexval} &= \text{?} \\
* \rightarrow & \quad F.\text{val} \quad T'.\text{lval} \quad T'.\text{val} \quad \text{?} \\
\epsilon \rightarrow & \quad T'.\text{val} \\
F.\text{val} & \quad \text{digit}.\text{lexval} \quad \text{?} \\
\text{digit}.\text{lexval} & \quad \epsilon
\end{align*}
\]
```
Example of Evaluation

\[
T \rightarrow F \ T'
\]
\[
T' \rightarrow \ast \ F \ T'
\]
\[
F \rightarrow \text{digit}
\]

\[
T'^{.lval} = F^{.val} \\
\text{head}^{.val} = T'^{.val} \\
T'^{.lval} = \text{head}^{.lval} \ast F^{.val} \\
\text{head}^{.val} = T'^{.val} \\
\text{head}^{.val} = \text{digit}^{.lexval}
\]

Input: 3 * 5

\[
\text{digit}^{.lexval}=? \\
\ast \\
F^{.val}=? \\
T'^{.lval}=? \\
T'^{.val}=? \\
\text{digit}^{.lexval}=? \\
\ast \\
F^{.val}=? \\
T'^{.lval}=? \\
T'^{.val}=? \\
\text{digit}^{.lexval}=? \\
\varepsilon
\]
Example of Evaluation

\[
\begin{align*}
T &\rightarrow F T' \\
T' &\rightarrow * F T' \\
F &\rightarrow \text{digit}
\end{align*}
\]

\[
\begin{align*}
\text{head.val} &= T'.val \\
T'.lval &= F.val \\
\text{head.val} &= T'.val \\
T'.lval &= \text{head.lval} * F.val \\
T'.lval &= \text{head.lval} \\
\text{head.val} &= \text{digit.lexval}
\end{align*}
\]

Input: 3 * 5

\[
\begin{align*}
T.val=? \\
F.val=? \\
\text{digit.lexval}=?
\end{align*}
\]

\[
\begin{align*}
T'.lval=? \\
T'.val=? \\
F.val=?
\end{align*}
\]

\[
\begin{align*}
T'.lval=? \\
T'.val=? \\
\epsilon
\end{align*}
\]
### Example of Evaluation

<table>
<thead>
<tr>
<th>Grammar Rule</th>
<th>Technique</th>
</tr>
</thead>
</table>
| $T \rightarrow F T'$ | $T'.lval = F.val$  
 $head.val = T'.val$ |
| $T' \rightarrow * F T'$ | $T'.lval = head.lval * F.val$  
 $head.val = T'.val$ |
| $F \rightarrow \epsilon$ | $head.val = head.lval$  
 $head.val = digit.lexval$ |
| $F \rightarrow digit$ | $head.val = digit.lexval$ |

**Input:** $3 \times 5$

```
T.val=?
F.val=?

* F.val=?

T'.lval=?
T'.val=?

digit.lexval=3

T'.lval= F.val
head.val = T'.val

head.val = T'.val
head.val = digit.lexval
```
Example of Evaluation

Input: 3 * 5

```
T  →  F T'
T' →  * F T'
F  →  digit

T'.lval = F.val
head.val = T'.val
T'.lval = head.lval * F.val
head.val = T'.val
head.val = head.lval
head.val = digit.lexval
```

```
T.val=?
F.val=3
T'.lval=?
T'.val=*
F.val=*
digit.lexval=*
T'.lval=*
T'.val=*
T'.lval=*
head.val = head.lval
T'.lval=*
T'.val=*
T'.lval=*
head.val = head.lval
head.val = digit.lexval
```
Example of Evaluation

\[
\begin{align*}
T & \rightarrow F T' \\
T' & \rightarrow * F T' \\
F & \rightarrow \text{digit}
\end{align*}
\]

Input: 3 * 5
Example of Evaluation

\[ T \rightarrow F \ T' \]
\[ T' \rightarrow * \ F \ T' \]
\[ F \rightarrow \text{digit} \]

\[ T'.lval = F.val \]
\[ \text{head.val} = T'.val \]
\[ T'.lval = \text{head.lval} * F.val \]
\[ \text{head.val} = T'.val \]
\[ \text{head.val} = \text{head.lval} \]
\[ \text{head.val} = \text{digit.lexval} \]

Input: 3 * 5
### Example of Evaluation

| $T$         | $F T'$                     | $T\'.lval = F.val$  
|            |                           | head.val = $T\'.val$  
| $T'$       | $* F T'$                  | $T\'.lval = head.lval \times F.val$  
|            | $|$ $\epsilon$            | head.val = $T\'.val$  
| $F$        | $\rightarrow$ $\rightarrow$ | head.val = head.lval  
|            | digit                      | head.val = digit.lexval |

**Input:**  $3 \times 5$

```
T.val=?

F.val=3

digit.lexval=3

*  

F.val=?  

T'.val=?  

T'.lval=3

digit.lexval=?

\epsilon
```
Example of Evaluation

\[
\begin{align*}
T & \rightarrow F T' \\
T' & \rightarrow \ast F T' \\
F & \rightarrow \text{digit}
\end{align*}
\]

Input: \(3 \ast 5\)

\[
\begin{align*}
T\text{.val} &= ? \\
F\text{.val} &= 3 \\
\text{digit}\text{.lexval} &= 3 \\
T'\text{.lval} &= 3 \\
T'\text{.val} &= ? \\
* & \rightarrow \\
F\text{.val} &= ? \\
T'\text{.lval} &= ? \\
T'\text{.val} &= ? \\
\text{digit}\text{.lexval} &= 5 \\
\varepsilon & \rightarrow
\end{align*}
\]
## Example of Evaluation

<table>
<thead>
<tr>
<th>Rule</th>
<th>Left-hand side</th>
<th>Right-hand side</th>
</tr>
</thead>
</table>
| $T$  | $F \ T'$      | $T'.lval = F.val$
|      |               | head.val = $T'.val$
|      | $T'$          | $T'.lval = head.lval * F.val$
|      | $*$ $F \ T'$ | head.val = $T'.val$
|      |               | head.val = head.lval
|      | $F$           | head.val = digit.lexval |

### Input: $3 \times 5$

```
T.val=?

F.val=3

digit.lexval=3

* F.val=5

T'.lval=?

T'.val=?

digit.lexval=5

ε
```
Example of Evaluation

Input: 3 * 5

\[
\begin{align*}
T & \rightarrow F T' \\
T' & \rightarrow * F T' \\
F & \rightarrow \text{digit}
\end{align*}
\]

\[
\begin{align*}
T'.lval &= F.val \\
\text{head}.val &= T'.val \\
T'.lval &= \text{head}.lval * F.val \\
\text{head}.val &= T'.val \\
\text{head}.val &= \text{head}.lval \\
\text{head}.val &= \text{digit}.\text{lexval}
\end{align*}
\]
Example of Evaluation

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Action</th>
</tr>
</thead>
</table>
| $T \rightarrow F T'$ | $T'.lval = F.val$
| $T' \rightarrow * F T'$ | $head.val = T'.val$
| | $T'.lval = head.lval * F.val$
| | $head.val = T'.val$
| | $head.val = head.lval$
| | $head.val = digit.lexval$

Input: $3 \times 5$

```
T.val=?
```
```
F.val=3
```
```
digit.lexval=3
```
```
* F.val=5
```
```
digit.lexval=5
```
```
T'.lval=15
```
```
T'.val=15
```
```
head.val = head.lval
```
```
\text{Input: } 3 \times 5
```

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
Example of Evaluation

\[
\begin{align*}
T & \rightarrow F T' \\
T' & \rightarrow * F T' \\
F & \rightarrow \text{digit}
\end{align*}
\]

Input: 3 * 5

\[
\begin{align*}
T.\text{val} & = \text{?} \\
F.\text{val} & = 3 \\
\text{digit}.\text{lexval} & = 3 \\
T'.\text{lval} & = \text{F.}\text{val} \\
\text{head.}\text{val} & = T'.\text{val} \\
T'.\text{lval} & = 15 \\
T'.\text{val} & = 15 \\
F.\text{val} & = 5 \\
\text{digit}.\text{lexval} & = 5 \\
* & \\
T'.\text{lval} & = \text{head.}\text{lval} \times F.\text{val} \\
\text{head.}\text{val} & = T'.\text{val} \\
T'.\text{lval} & = 15 \\
T'.\text{val} & = 15 \\
\epsilon & \\
\end{align*}
\]
Example of Evaluation

\[
\begin{align*}
T & \rightarrow F \ T' \\
T' & \rightarrow * \ F \ T' \\
F & \rightarrow \ digit
\end{align*}
\]

Input: \(3 \times 5\)

\[
\begin{align*}
T'.lval &= F.val \\
head.val &= T'.val \\
T'.lval &= head.lval \times F.val \\
head.val &= T'.val \\
head.val &= head.lval \\
head.val &= digit.lexval
\end{align*}
\]

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
Problem of the Evaluation Order

How to determine the correct sequence of evaluations of the semantic rules’ lines?

Introduction of a graph of dependencies between the attributes.
Outline

1 Introduction

2 Translation scheme
   - Syntax-directed translation
   - Syntax-directed definition
   - Attributes of the productions
   - Evaluating a SDD with a parse tree
   - Dependency graph
     - S-attributed definition
     - L-attributed definition

3 Syntax tree and graph

4 Three-address code

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
Definition

A dependency graph depicts the flow of information among the attribute instances in a particular parse tree.

- **Node**: For each parse-tree node, say a node labeled by grammar symbol $X$, the dependency graph has a node for each attribute associated with $X$.

- **Edge**: Between two attribute instances. Means that the value of the first is needed to compute the second.
Examples of Atomic Dependency Graphs

\[
\begin{align*}
T & \rightarrow F T' & T'.lval &= F.val \\
T' & \rightarrow * F T' & head.val &= T'.val \\
& \mid & T'.lval &= head.lval * F.val \\
F & \rightarrow \text{digit} & head.val &= T'.val \\
& & head.val &= \text{digit}.lexval
\end{align*}
\]
Examples of Atomic Dependency Graphs

\[
\begin{align*}
T & \rightarrow \ F \ T' \\
T' & \rightarrow \ * \ F \ T' \\
F & \rightarrow \ \text{digit}
\end{align*}
\]

\[
\begin{align*}
T'.lval &= F.val \\
\text{head.val} &= T'.val \\
T'.lval &= \text{head.lval} * F.val \\
\text{head.val} &= T'.val \\
\text{head.val} &= \text{head.lval} \\
\text{head.val} &= \text{digit.lexval}
\end{align*}
\]
Examples of Atomic Dependency Graphs

\[
\begin{align*}
T & \rightarrow F T' \\
T' & \rightarrow * F T' \\
F & \rightarrow \text{digit}
\end{align*}
\]

\[
\begin{align*}
T'.lval &= F.val \\
\text{head.val} &= T'.val \\
T'.lval &= \text{head.lval} * F.val \\
\text{head.val} &= T'.val \\
\text{head.val} &= \text{head.lval} \\
\text{head.val} &= \text{digit.lexval}
\end{align*}
\]
Examples of Atomic Dependency Graphs

\[
\begin{align*}
T & \rightarrow F T' \\
T' & \rightarrow * F T' \\
F & \rightarrow \text{digit}
\end{align*}
\]

\[
\begin{align*}
T'.\text{lval} &= F.\text{val} \\
\text{head.}\text{val} &= T'.\text{val} \\
T'.\text{lval} &= \text{head.}\text{lval} * F.\text{val} \\
\text{head.}\text{val} &= T'.\text{val} \\
\text{head.}\text{val} &= \text{head.}\text{lval} \\
\text{head.}\text{val} &= \text{digit.}\text{lexval}
\end{align*}
\]
Examples of Atomic Dependency Graphs

<table>
<thead>
<tr>
<th>Production</th>
<th>Action</th>
</tr>
</thead>
</table>
| $T \rightarrow FT'$ | $T'.lval = F.val$
|             | head.val = $T'.val$     |
| $T' \rightarrow *FT'$ | $T'.lval = head.lval * F.val$
|             | head.val = $T'.val$     |
|             | $\epsilon$              |
| $F \rightarrow \text{digit}$ | head.val = head.lval   |
|             | head.val = $\text{digit}.lexval$ |

$T_{\text{val}}$ | $F_{\text{val}}$ | $T'_{lval}$ | $T_{lval}$ | $F_{\text{val}}$ | $T'_{lval}$ | $\epsilon$ | $\text{digit}$
Examples of Dependency Graph on Input

\[
\begin{align*}
T & \rightarrow F T' \\
T' & \rightarrow * F T' \\
F & \rightarrow \text{digit}
\end{align*}
\]

\[
\begin{align*}
T'.lval &= F.val \\
\text{head.val} &= T'.val \\
T'.lval &= \text{head.lval} * F.val \\
\text{head.val} &= T'.val \\
| \epsilon & \\
\text{head.val} &= \text{head.lval} \\
\text{head.val} &= \text{digit}.lexval
\end{align*}
\]

Input: 3 * 5
The dependency graph characterizes the possible orders in which we can evaluate the attributes at the various nodes of a parse tree.

If the dependency graph has an edge from node $M$ to node $N$, then the attribute corresponding to $M$ must be evaluated before the attribute of $N$.

The only allowable orders of evaluation are those sequences of nodes $N_1, N_2, \ldots, N_k$ such that if there is an edge of the dependency graph from $N_i$ to $N_j$, then $i < j$.

Such an ordering embeds a directed graph into a linear order, and is called a topological sort of the graph.
If there is any cycle in the graph, then there are no topological sorts; i.e., there is no way to evaluate the SDD on the parse tree.

Given a SDD, it is very hard to tell whether there exist any parse trees whose dependency graphs have cycles.

- Translations can be implemented using classes of SDD that guarantee an evaluation order, i.e., without cycle.

- Two possible approaches to solve this problem:
  1. S-attributed definition.
  2. L-attributed definition.
Outline

1 Introduction

2 Translation scheme
   - Syntax-directed translation
   - Syntax-directed definition
   - Attributes of the productions
   - Evaluating a SDD with a parse tree
   - Dependency graph
   - S-attributed definition
   - L-attributed definition

3 Syntax tree and graph

4 Three-address code
An S-attributed definition is an SDD in which all the attributes are synthesized.

- S-attributed definitions can be implemented during bottom-up parsing, since a bottom-up parse corresponds to a postorder traversal of the parse tree.
Outline

1 Introduction

2 Translation scheme
   - Syntax-directed translation
   - Syntax-directed definition
   - Attributes of the productions
   - Evaluating a SDD with a parse tree
   - Dependency graph
   - S-attributed definition
   - L-attributed definition

3 Syntax tree and graph

4 Three-address code
An L-attributed definition is an SDD in which, between the attributes associated with a production body, dependency-graph edges can go from left to right, but not from right to left.

Each attribute must be:

- Synthesized, or
- Inherited, with the rules limited as follows.

Suppose a production $A \rightarrow X_1 X_2 \ldots X_n$, and an inherited attribute $X_i.a$. The rule may use only:

a) Inherited attributes associated with the head $A$.

b) Either inherited or synthesized attributes associated with the occurrences of symbols $X_1 X_2 \ldots X_{i-1}$ located to the left of $X_i$.

c) Inherited or synthesized attributes associated with this occurrence of $X_i$ itself, but only in such a way that there are no cycles in a graph dependency formed by the attributes of this $X$. 
Outline

1 Introduction

2 Translation scheme

3 Syntax tree and graph
   - Syntax tree
     - Definition
     - Building from S-attributed definition
     - Building from L-attributed definition
   - Directed acyclic graph

4 Three-address code

5 Code generation of variables
Since some compilers use syntax tree as an intermediate representation, a common form of SDD turns its input string into a tree.

To complete the translation to intermediate code, the compiler may then walk the syntax tree, using another set of rules that are in effect an SDD on the syntax tree rather than the parse tree.

Two SDD are considered in this section to build a syntax tree:

- S-attributed definition (bottom-up), and
- L-attributed definition (top-down).
Each node in a syntax tree represents a language construct.

The children of the node represent the meaningful components of the construct.

We shall implement the nodes of a syntax tree by objects with a suitable number of fields.

Each object will have an operator field that is the label of the node, and the following additional fields:

- If the node is a leaf, the lexical value for the leaf.
- If the node is an interior node, all the children are stored in individual fields.
Example of a Syntax Tree

This is the syntax tree for the statement:

\[ a := 3 + (6 \times 7) \]
1 Introduction

2 Translation scheme

3 Syntax tree and graph
   - Syntax tree
     - Definition
     - Building from S-attributed definition
     - Building from L-attributed definition
   - Directed acyclic graph

4 Three-address code

5 Code generation of variables
### S-Attributed Definition for Building a Syntax Tree

<table>
<thead>
<tr>
<th>$E$</th>
<th>$E + T$</th>
<th>head.node = <code>new</code> Node(&quot;+&quot;, E.node, T.node)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$E - T$</td>
<td>head.node = <code>new</code> Node(&quot;-&quot;, E.node, T.node)</td>
</tr>
<tr>
<td></td>
<td>$T$</td>
<td>head.node = T.node</td>
</tr>
<tr>
<td>$T$</td>
<td>$(E)$</td>
<td>head.node = E.node</td>
</tr>
<tr>
<td></td>
<td>id</td>
<td>head.node = <code>new</code> Leaf(id, id.lexeme)</td>
</tr>
<tr>
<td></td>
<td>num</td>
<td>head.node = <code>new</code> Leaf(num, num.value)</td>
</tr>
</tbody>
</table>

- The semantic rules contain the creation of the syntax tree nodes, with the description of the node and/or the children as parameters. The root of the syntax tree becomes $E$.node.

- Note that the annotated parse tree is implicitly defined by the grammar rules and not directly built by the compiler.
Example of Syntax Tree Building

$E \rightarrow E + T$
$|\quad E - T$
$|\quad T$

$T \rightarrow (E)$
$|\quad id$
$|\quad num$

Input: $a - 4 + c$

```
E node
  E node
    E node
      T node
        id
      num
    - T node
      id
  + T node
```
Example of Syntax Tree Building

\[
E \rightarrow E + T \\
| E - T \\
| T
\]

\[
T \rightarrow (E) \\
| id \\
| num
\]

Input: \(a - 4 + c\)
Example of Syntax Tree Building

**Input:** \( a - 4 + c \)

**Example of Syntax Tree Building**

```
E → E + T
   | E - T
   | T
T → ( E )
   | id
   | num
```

- \( E \) node
- \( E \) node
- \( E \) node
- \( T \) node
- \( T \) node
- \( T \) node
- \( id \)
- \( id \)
- \( num \)

\[
\begin{align*}
\text{head.node} & = \textbf{new} \ \text{Node}("\+", \text{E.node, T.node}) \\
\text{head.node} & = \textbf{new} \ \text{Node}("-", \text{E.node, T.node}) \\
\text{head.node} & = \text{T.node} \\
\text{head.node} & = \text{E.node} \\
\text{head.node} & = \textbf{new} \ \text{Leaf}({\text{id, id.lexeme}}) \\
\text{head.node} & = \textbf{new} \ \text{Leaf}({\text{num, num.value}})
\end{align*}
\]
### Example of Syntax Tree Building

**Input:** $a - 4 + c$

<table>
<thead>
<tr>
<th>$E$</th>
<th>$E + T$</th>
<th>head.node = new Node(&quot;+&quot;, E.node, T.node)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E$</td>
<td>$E - T$</td>
<td>head.node = new Node(&quot;-&quot;, E.node, T.node)</td>
</tr>
<tr>
<td>$T$</td>
<td></td>
<td>head.node = T.node</td>
</tr>
<tr>
<td>$T$</td>
<td>$(E)$</td>
<td>head.node = E.node</td>
</tr>
<tr>
<td></td>
<td>$id$</td>
<td>head.node = new Leaf(id, id.lexeme)</td>
</tr>
<tr>
<td></td>
<td>$num$</td>
<td>head.node = new Leaf(num, num.value)</td>
</tr>
</tbody>
</table>

**Diagram:**

- **$E$ node**
  - **$E$ node** - $+$ - **$T$ node**
  - **$E$ node** - $-$ - **$T$ node** $id$
  - **$T$ node** $num$

- **id** in symbol table

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
Example of Syntax Tree Building

\[
E \rightarrow E + T \\
| E - T \\
| T
\]

\[
T \rightarrow ( E ) \\
| id \\
| num
\]

Input: \( a - 4 + c \)
Example of Syntax Tree Building

\[
\begin{align*}
E &\rightarrow E + T \\
&\quad | E - T \\
&\quad | T \\
T &\rightarrow (E) \\
&\quad | \text{id} \\
&\quad | \text{num}
\end{align*}
\]

\[
\begin{align*}
\text{head.node} &= \textbf{new} \ \text{Node}(" +", \ E.\text{node}, \ T.\text{node}) \\
\text{head.node} &= \textbf{new} \ \text{Node}(" -", \ E.\text{node}, \ T.\text{node}) \\
\text{head.node} &= \ T.\text{node} \\
\text{head.node} &= \ E.\text{node} \\
\text{head.node} &= \textbf{new} \ \text{Leaf}(\text{id}, \ \text{id}.\text{lexeme}) \\
\text{head.node} &= \textbf{new} \ \text{Leaf}(\text{num}, \ \text{num}.\text{value})
\end{align*}
\]

Input: \( a - 4 + c \)

\( a \) in symbol table

\[\text{id} \quad \text{num} 4\]
### Example of Syntax Tree Building

**Production Rules:**

<table>
<thead>
<tr>
<th>$E$</th>
<th>$E + T$</th>
<th>$E - T$</th>
<th>$T$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$E + T$</td>
<td>$E - T$</td>
<td>$T$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$(E)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$id$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$num$</td>
</tr>
</tbody>
</table>

**Tree Construction:**

- **Input:** $a - 4 + c$
- **Head Node:**
  - Non-terminal $E$
- **Left Child:**
  - Non-terminal $E$
  - Value: $a - 4 + c$
  - Symbol Table: $a$
- **Right Child:**
  - Non-terminal $T$
  - Value: $4$

**Syntax Tree:**

```
E node
  E node
    E node
      $a$
      -
      T node
        +
        T node
          $-$
          id
          num
          $4$
```

**Generated Code:**

```java
head.node = new Node("+", E.node, T.node)
head.node = new Node("-", E.node, T.node)
head.node = T.node
head.node = E.node
head.node = new Leaf(id, id.lexeme)
head.node = new Leaf(num, num.value)
```
Example of Syntax Tree Building

**Input:** a - 4 + c

<table>
<thead>
<tr>
<th>$E$</th>
<th>$E + T$</th>
<th>head.node = new Node(&quot;+&quot;, E.node, T.node)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$E - T$</td>
<td>head.node = new Node(&quot;-&quot;, E.node, T.node)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>head.node = T.node</td>
</tr>
<tr>
<td>$T$</td>
<td>$(E)$</td>
<td>head.node = E.node</td>
</tr>
<tr>
<td></td>
<td>id</td>
<td>head.node = new Leaf(id, id.lexeme)</td>
</tr>
<tr>
<td></td>
<td>num</td>
<td>head.node = new Leaf(num, num.value)</td>
</tr>
</tbody>
</table>

```
head.node = new Node("+", E.node, T.node)
```
Example of Syntax Tree Building

Input: \( a - 4 + c \)

\[
\begin{align*}
E & \rightarrow E + T \\
& \mid E - T \\
& \mid T \\
T & \rightarrow (E) \\
& \mid id \\
& \mid num
\end{align*}
\]

- head.node = \texttt{new Node("+", E.node, T.node)}
- head.node = \texttt{new Node("-", E.node, T.node)}
- head.node = T.node
- head.node = E.node
- head.node = \texttt{new Leaf(id, id.lexeme)}
- head.node = \texttt{new Leaf(num, num.value)}

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
Outline

1 Introduction

2 Translation scheme

3 Syntax tree and graph
   - Syntax tree
     - Definition
     - Building from S-attributed definition
     - Building from L-attributed definition
   - Directed acyclic graph

4 Three-address code

5 Code generation of variables

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
L-Attributed Definition for Building a Syntax Tree

<table>
<thead>
<tr>
<th>$E$</th>
<th>$TE'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E'$</td>
<td>$T E'$</td>
</tr>
<tr>
<td>$E'$</td>
<td>$+T E'$</td>
</tr>
<tr>
<td>$E'$</td>
<td>$-T E'$</td>
</tr>
<tr>
<td>$T$</td>
<td>$(E)$</td>
</tr>
<tr>
<td></td>
<td>$id$</td>
</tr>
<tr>
<td></td>
<td>$num$</td>
</tr>
</tbody>
</table>

Note that the annotated parse tree is implicitly defined by the grammar rules and not directly built by the compiler.
Example of Syntax Tree Building

The syntax tree for the expression `a - 4 + c` is as follows:

```
E -> T E'
E' -> + T E' | - T E' | ε
T -> ( E ) | id | num
```

```
E'.left = T.node
head.node = E'.node
E'.parent = new Node("+", head.left, T.node)
head.node = E'.node
E'.parent = new Node("-", head.left, T.node)
head.node = E'.node
head.node = head.left
head.node = E.node
head.node = new Leaf(id, id.lexeme)
head.node = new Leaf(num, num.value)
```

Input: `a - 4 + c`
Example of Syntax Tree Building

\[
\begin{align*}
E & \rightarrow \quad T \ E' \\
E' & \rightarrow \quad + \ T \ E' \\
& \quad | \quad - \ T \ E' \\
& \quad | \quad \epsilon \\
T & \rightarrow \quad ( \ E ) \\
& \quad | \quad id \\
& \quad | \quad num
\end{align*}
\]

\[
\begin{align*}
E'.left &= T.node \\
head.node &= E'.node \\
E'.parent &= \text{new Node}("\+", \text{head.left}, T.node) \\
head.node &= E'.node \\
E'.parent &= \text{new Node}("-", \text{head.left}, T.node) \\
head.node &= E'.node \\
head.node &= \text{head.left} \\
T.node &= E.node \\
head.node &= \text{new Leaf(id, id.lexeme)} \\
head.node &= \text{new Leaf(num, num.value)}
\end{align*}
\]

Input: a - 4 + c
### Example of Syntax Tree Building

<table>
<thead>
<tr>
<th>Rule</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E \to T E'$</td>
<td>$E'\text{.left} = T\text{.node}$&lt;br&gt;$\text{head.node} = E'\text{.node}$</td>
</tr>
<tr>
<td>$E' \to + T E'$</td>
<td>$E'\text{.parent} = \text{new Node}(&quot;+&quot;, \text{head.left}, T\text{.node})$&lt;br&gt;$\text{head.node} = E'\text{.node}$</td>
</tr>
<tr>
<td>$E' \to - T E'$</td>
<td>$E'\text{.parent} = \text{new Node}(&quot;-&quot;, \text{head.left}, T\text{.node})$&lt;br&gt;$\text{head.node} = E'\text{.node}$</td>
</tr>
<tr>
<td>$E' \to \epsilon$</td>
<td>$\text{head.node} = \text{head.left}$</td>
</tr>
<tr>
<td>$T \to (E)$</td>
<td>$\text{head.node} = E\text{.node}$</td>
</tr>
<tr>
<td>$T \to id$</td>
<td>$\text{head.node} = \text{new Leaf}(id, id\text{.lexeme})$</td>
</tr>
<tr>
<td>$T \to num$</td>
<td>$\text{head.node} = \text{new Leaf}(num, num\text{.value})$</td>
</tr>
</tbody>
</table>

#### Input: $a - 4 + c$

```
E → T E'
E' → + T E'
E' → - T E'
E' → \epsilon
T → (E)
T → id
T → num
```

![Syntax Tree Diagram](image-url)
Example of Syntax Tree Building

$$E \rightarrow \ T \ E'$$
E'.left = T.node
head.node = E'.node

$$E' \rightarrow \ + \ T \ E'$$
E'.parent = new Node("+", head.left, T.node)
head.node = E'.node

$$| - \ T \ E'$$
E'.parent = new Node("-", head.left, T.node)
head.node = E'.node

$$| \ \epsilon$$
head.node = head.left

$$T \rightarrow \ ( \ E )$$
head.node = E.node

$$| \ id$$
head.node = new Leaf(id, id.lexeme)

$$| \ num$$
head.node = new Leaf(num, num.value)

Input: a - 4 + c
Example of Syntax Tree Building

| E  | → | T E' | E'.left = T.node  
|    |   |      | head.node = E'.node  
| E' | → | + T E' | E'.parent = new Node("+", head.left, T.node)  
|    |   |      | head.node = E'.node  
|    | | - T E' | E'.parent = new Node("-", head.left, T.node)  
|    |   |      | head.node = E'.node  
|    | | ε   | head.node = head.left  
| T  | → | ( E ) | head.node = E.node  
|    | | id  | head.node = new Leaf(id, id.lexeme)  
|    | | num | head.node = new Leaf(num, num.value)  

Input: a - 4 + c

Example of Syntax Tree Building:

```
E  →  T E'  E'.left = T.node  
    →  + T E' E'.parent = new Node("+", head.left, T.node)  
        |    | - T E' E'.parent = new Node("-", head.left, T.node)  
        |    | ε head.node = head.left  
T  →  ( E ) head.node = E.node  
    →  id head.node = new Leaf(id, id.lexeme)  
    →  num head.node = new Leaf(num, num.value)  
```

Diagram:

```
E node 13  
   T node 12  
      E node left 3  
         E node left 11  
            T node 6  
                E node left 9  
                   T node 4  
                       T node 5  
                           T node 2  
                               T node 1  
                                   id 1  
                                      -  
                                          T node 5  
                                             T node 3  
                                                E node left 10  
                                                   E node left 8  
                                                      T node 7  
                                                          T node 1  
                                                              id 1  
```

Example of Syntax Tree Building:

```
a in symbol table
```
Example of Syntax Tree Building

\[
E \rightarrow T E' \\
E' \rightarrow + T E' \\
| \rightarrow - T E' \\
| \rightarrow \epsilon \\
T \rightarrow ( E ) \\
| \rightarrow id \\
| \rightarrow num
\]

\[
E'.left = T.node \\
head.node = E'.node \\
E'.parent = \text{new} \ Node(" + ", head.left, T.node) \\
head.node = E'.node \\
E'.parent = \text{new} \ Node(" - ", head.left, T.node) \\
head.node = E'.node \\
head.node = \epsilon \\
head.node = head.left \\
head.node = E.node \\
head.node = \text{new} \ \text{Leaf}(id, id.lexeme) \\
head.node = \text{new} \ \text{Leaf}(num, num.value)
\]

Input: \( a - 4 + c \)
Example of Syntax Tree Building

\[
\begin{align*}
E & \rightarrow T E' \\
E' & \rightarrow + T E' \\
& \mid - T E' \\
& \mid \epsilon \\
T & \rightarrow ( E ) \\
& \mid id \\
& \mid num
\end{align*}
\]

Input: \( a - 4 + c \)
Example of Syntax Tree Building

\[

e \rightarrow T E' \\
E' \rightarrow + T E' \\
| - T E' \\
| \epsilon \\
T \rightarrow ( E ) \\
| id \\
| num
\]

E'.left = T.node
head.node = E'.node
E'.parent = new Node("+", head.left, T.node)
head.node = E'.node
E'.parent = new Node("-", head.left, T.node)
head.node = E'.node
head.node = head.left
head.node = E.node
head.node = new Leaf(id, id.lexeme)
head.node = new Leaf(num, num.value)

Input: a - 4 + c
Example of Syntax Tree Building

\[
\begin{align*}
E & \rightarrow T E' \\
E' & \rightarrow + T E' \\
& \quad \mid - T E' \\
T & \rightarrow ( E ) \\
& \quad \mid id \\
& \quad \mid num
\end{align*}
\]

\[
\begin{align*}
E'.left &= T.node \\
head.node &= E'.node \\
E'.parent &= new \text{Node}("+", head.left, T.node) \\
head.node &= E'.node \\
E'.parent &= new \text{Node}("-", head.left, T.node) \\
head.node &= E'.node \\
head.node &= head.left \\
head.node &= E.node \\
head.node &= new \text{Leaf}(id, id.lexeme) \\
head.node &= new \text{Leaf}(num, num.value)
\end{align*}
\]

Input: \(a - 4 + c\)
Example of Syntax Tree Building

```
E  →  T E'          E'.left = T.node
E' →  + T E'        head.node = E'.node
    | - T E'          E'.parent = new Node("+", head.left, T.node)
    |      ∈           head.node = E'.node
T  →  ( E )         E'.parent = new Node("-", head.left, T.node)
    | id             head.node = E.node
    | num            head.node = new Leaf(id, id.lexeme)
                  head.node = new Leaf(num, num.value)
```

Input: a - 4 + c
Example of Syntax Tree Building

\[
\begin{align*}
E & \rightarrow T E' \\
E' & \rightarrow + T E' \\
& \mid - T E' \\
& \mid \epsilon \\
T & \rightarrow (E) \\
& \mid id \\
& \mid num
\end{align*}
\]

\[
E'.left = T.node \\
head.node = E'.node
\]

\[
E'.parent = new Node("+", head.left, T.node) \\
head.node = E'.node
\]

\[
E'.parent = new Node("-", head.left, T.node) \\
head.node = E'.node
\]

\[
head.node = head.left
\]

\[
head.node = E.node
\]

\[
head.node = new Leaf(id, id.lexeme)
\]

\[
head.node = new Leaf(num, num.value)
\]

Input: \(a - 4 + c\)
Example of Syntax Tree Building

<table>
<thead>
<tr>
<th>Production</th>
<th>Syntax Tree Construction</th>
</tr>
</thead>
</table>
| $E \rightarrow TE'$ | $E'.left = T.node$  
head.node = $E'.node$  
E'.parent = new Node("+", head.left, T.node)  
head.node = $E'.node$  
|
| $E' \rightarrow + TE'$ | $E'.parent = new Node("+", head.left, T.node)$  
head.node = $E'.node$  
head.node = $E'.node$  
E'.parent = new Node("+", head.left, T.node)  
head.node = $E'.node$  
|
| $E' \rightarrow - TE'$ | $E'.parent = new Node("-", head.left, T.node)$  
head.node = $E'.node$  
head.node = $E'.node$  
head.node = head.left  
|
| $E' \rightarrow \epsilon$ | head.node = head.left  
|
| $T \rightarrow (E)$ | head.node = $E.node$  
head.node = new Leaf(id, id.lexeme)  
|
| $T \rightarrow id$ | head.node = new Leaf(id, id.lexeme)  
|
| $T \rightarrow num$ | head.node = new Leaf(num, num.value)  
|

Input: $a - 4 + c$
Example of Syntax Tree Building

\[
\begin{align*}
E & \rightarrow T E' \\
E' & \rightarrow + T E' \\
& \mid - T E' \\
& \mid \varepsilon \\
T & \rightarrow ( E ) \\
& \mid \text{id} \\
& \mid \text{num}
\end{align*}
\]

\[
E'.\text{left} = T.\text{node} \\
\text{head.\text{node}} = E'.\text{node}
\]

\[
E'.\text{parent} = \text{new Node(“+”, head.\text{left}, T.\text{node})} \\
\text{head.\text{node}} = E'.\text{node}
\]

\[
E'.\text{parent} = \text{new Node(“-”, head.\text{left}, T.\text{node})} \\
\text{head.\text{node}} = E'.\text{node}
\]

\[
T.\text{left} = T.\text{node} \\
\text{head.\text{node}} = \text{head.\text{left}}
\]

\[
T.\text{node} = ( E ) \\
\mid \text{id} \\
\mid \text{num}
\]

\[
\text{id in symbol table} \\
\text{num 4}
\]

Input: \( a - 4 + c \)
Outline

1. Introduction
2. Translation scheme
3. Syntax tree and graph
   - Syntax tree
   - Directed acyclic graph
4. Three-address code
5. Code generation of variables
6. Code generation of statements
Nodes in a syntax tree represent language constructs in the source program; the children of a node represent the meaningful components of a construct.

Definition (Directed Acyclic Graph)

A directed acyclic graph (DAG) represent the language constructs in the source program. It ensures that a construct is present only one time in the DAG.

The difference between syntax tree and DAG is that the DAG node may have more than one parent.

Consequently, a subexpression is repeated in a tree; and shared in a DAG.
Example of Directed Acyclic Graph

\[ a + a^* (b - c) + (b - c)^* d \]
Example of Directed Acyclic Graph

\[ a + a^* (b - c) + (b - c)^* d \]

Same subexpressions are duplicated in the tree

Syntax Tree
Example of Directed Acyclic Graph

\[ a + a^* (b - c) + (b - c)^* d \]

Syntax Tree

DAG

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
Value-Number Representation for the DAG

- Often, the nodes of a DAG are stored in an array of records (also true for a syntax tree).

```
struct {
    int token_id;
    union {
        unsigned int symbol_index;
        double fvalue;
        long lvalue;
    }
    struct {
        unsigned int left;
        unsigned int right;
    } operands;
    } attr;
} Record;
```

- Each node of the DAG is referred by its index in the table; its value number.
- Let the signature of an interior node be the triple \( \langle op, l, r \rangle \), where \( op \) is the label, \( l \) its left child’s value number, and \( r \) its right child’s value number. \( l \) and \( r \) are set to 0 when there is no child.
Building a DAG

■ **INPUT:** Label \( op \), node \( l \), and node \( r \).

■ **OUTPUT:** The value number of a node in the array with signature \( \langle op, l, r \rangle \).

■ **METHOD:** Search the array for a node \( M \) with label \( op \), left child \( l \), and right child \( r \). If there is such a node return the value number of \( M \). If not, create in the array a new node \( N \) with label \( op \), left child \( l \), right child \( r \), and return its value number.

■ **NOTE:** This approach is searching the entire array every time; it is not efficient. A more efficient approach is to use a hash table, in which the nodes are put into “buckets,” each of which typically will have only few nodes.
Outline

1 Introduction
2 Translation scheme
3 Syntax tree and graph
4 Three-address code
   - Language basics
   - Quadruple form
   - Triple form
   - Static single-assignment form
5 Code generation of variables

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
The syntax tree is a **high-level** intermediate representation of the source program.

To make easier the generation of low-level code, a low-level intermediate representation is useful.

The **three-address code** is a form of low-level intermediate representation that is close to the assembler languages.

**Note:** The rest of this lecture focuses on the generation of code with three-address code. The syntax tree may be also used as a basis of the generation (see the supervised tutorials).
In three-address code, there is at most one operator on the right side of an instruction.

\[
\begin{align*}
t_1 &= y \ast z \\
t_2 &= x + t_2
\end{align*}
\]

where \( t_1 \) and \( t_2 \) are compiler-generated temporary names.
An address can be one of the following:

1. **Name**: For convenience, we allow source-program names to appear as addresses in three-address code. In an implementation, a source name is replaced by a pointer to its symbol-table entry, where all information about the name is kept.

2. **Constant**: A compiler must deal with many different types of constants and variables.

3. **Compiler-generated temporary Name**: It is useful, especially in optimizing compilers, to create a distinct name each time a temporary is needed. These temporaries can be combined, if possible, when registers are allocated to variables.
A symbolic label represents the index of a three-address instruction in the sequence of instructions.

Numeric positions can be substituted for the labels, either by making a separate pass or by “backpatching.”

Symbolic Label

L: \( t_1 = i + 1 \)
\( i = t_1 \)
\( t_2 = i \times 8 \)
\( t_3 = a [ t_2 ] \)
if \( t_3 < v \) then goto L

Numeric Position

103: \( t_1 = i + 1 \)
104: \( i = t_1 \)
105: \( t_2 = i \times 8 \)
106: \( t_3 = a [ t_2 ] \)
107: if \( t_3 < v \) then goto 103
Assignment instruction of the form \( x = y \ <op> \ z \), where \(<op>\) is a binary arithmetic or logical operation, and \(x\), \(y\), and \(z\) are addresses.

Assignment instruction of the form \( x = \ <op> \ y \), where \(<op>\) is an unary operation. Essential unary operations include unary minus, logical negation, and conversion operators.

Copy instruction of the form \( x = y \), where \(x\) is assigned the value of \(y\).
Jumping Instruction

- An unconditional jump `goto L`. The three-address instruction with label `L` is the next executed.

- Conditional jumps of the form `if x goto L`, and `ifFalse x goto L`. These instructions execute the instruction with label `L` next if `x` is true and false, respectively. Otherwise, the following three-address instruction in sequence is executed next, as usual.

- Conditional jumps such as `if x <relop> y goto L`, which apply a relational operator (`<`, `<=`, `>`...) to `x` and `y`, and execute the instruction with label `L` next if `x` stands in relation `<relop>` to `y`. If not, the following three-address instruction in sequence is executed next.
Procedure calls and returns are implemented using the following instructions:

- **param** \( x \) for passing values as parameters
- **call** \( p,n \) for the procedure call
- \( y = \text{call} \ p,n \) for the function call
- **return** \( y \) for returning a value.

where \( x \) and \( y \) are addresses, \( p \) is the name of the subroutine, \( n \) is the number of parameters to pass to the subroutine.

Procedures and their implementation are detailed in Chapter 548.
Indexed copy instruction has one of the forms:

- $x = y[i]$
- $x[i] = y$

The first instruction form sets $x$ to the value of the $i^{th}$ memory unit beyond $y$.

The second instruction form sets the contents of the $i^{th}$ unit beyond $x$ to the value of $y$. 
Address and pointer assignment has one of the forms:

- \( x = \& y \): sets \( x \) to the location of \( y \).

- \( x = \ast y \): sets \( x \) to the value of the variable at the memory address \( y \).

- \( \ast x = y \): sets the variable at the memory address \( x \) to the value of \( y \).
Consider the statement:

```plaintext
do i = i + 1; while (a[i] < v);
```

The translation is three-address code is the following, assuming that the type of the elements of "a" takes 8 units of memory each.

```
L: t_1 = i + 1
    i = t_1
    t_2 = i * 8
    t_3 = a[t_1]
    if t_2 < v then goto L
```
The choice of allowable operators is an important issue in the design of an intermediate form.

The operator set clearly must be rich enough to implement the operations from the source language.

Operators that are close to machine instructions make it easier to implement the intermediate form on a target machine.

However, if the front end must generate long sequences of instructions for some source-language operations, then the optimizer and code generator may have to work harder to rediscover the structure and generate good code for these operations.
Three-address instructions could be implemented in a compiler as objects or as records.

Three representations are commonly used: quadruple, triples, and indirect triples.
1 Introduction

2 Translation scheme

3 Syntax tree and graph

4 Three-address code
  - Language basics
  - Quadruple form
  - Triple form
  - Static single-assignment form

5 Code generation of variables
A quadruple has four fields:

\[
<\text{op}> \ <\text{arg}_1> \ <\text{arg}_2> \ <\text{result}>
\]

- \(<\text{op}>\): this field contains an internal code for the operator.
- \(<\text{arg}_1>\) and \(<\text{arg}_2>\): they are the arguments of the operator.
- \(<\text{result}>\): it contains the result value computed by the operator.
Example of Quadruples

<table>
<thead>
<tr>
<th>op</th>
<th>arg₁</th>
<th>arg₂</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>minus</td>
<td>c</td>
<td></td>
<td>t₁</td>
</tr>
<tr>
<td>*</td>
<td>b</td>
<td>t₁</td>
<td>t₂</td>
</tr>
<tr>
<td>minus</td>
<td>c</td>
<td></td>
<td>t₃</td>
</tr>
<tr>
<td>*</td>
<td>b</td>
<td>t₃</td>
<td>t₄</td>
</tr>
<tr>
<td>+</td>
<td>t₂</td>
<td>t₄</td>
<td>t₅</td>
</tr>
<tr>
<td>=</td>
<td>t₅</td>
<td></td>
<td>a</td>
</tr>
</tbody>
</table>

\[ t₁ = \text{minus} \ c \\\n t₂ = b \times t₁ \\\n t₃ = \text{minus} \ c \\\n t₄ = b \times t₃ \\\n a = t₅ \]

\[ t₁ = \text{minus} \ c \]
\[ t₂ = b \times t₁ \]
\[ t₃ = \text{minus} \ c \]
\[ t₄ = b \times t₃ \]
\[ a = t₅ \]
Example of Implementation in C

```c
/* Operations supported by the three-address code */
typedef enum { MULTIPLY, ADD, MINUS, ... } Operator;

/* Definition of a parameter or a return value */
typedef union {
    unsigned long address; /* address of a variable */
    long integer_value;   /* integer constant */
    double float_value;   /* floating-point constant */
} Value;

/* Definition of a single quadruple */
typedef struct {
    Operator operator;
    Value arg1;
    Value arg2;
    Value result;
} Quadruple;
```
1 Introduction

2 Translation scheme

3 Syntax tree and graph

4 Three-address code
   - Language basics
   - Quadruple form
   - Triple form
   - Static single-assignment form

5 Code generation of variables
A triple has only three fields:

\(<op> <arg_1> <arg_2>\)

Note that result in quadruples is primarily used for temporary names.

The result of an operation \(x <op> y\) is referred by its position, rather than by an explicit temporary name.

Thus instead of the temporary \(t_1\) in the previous example, a triple representation would refer to position \((0)\).

Parenthesized numbers represent points into the triple structure itself.
Example of Triples

\[
\begin{align*}
t_1 &= \text{minus } c \\
t_2 &= b \times t_1 \\
t_3 &= \text{minus } c \\
t_4 &= b \times t_3 \\
t_5 &= t_2 + t_4 \\
a &= t_5
\end{align*}
\]

<table>
<thead>
<tr>
<th>op</th>
<th>arg1</th>
<th>arg2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>minus</td>
<td>c</td>
</tr>
<tr>
<td>1</td>
<td>\times</td>
<td>b</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(0)</td>
</tr>
<tr>
<td>2</td>
<td>minus</td>
<td>c</td>
</tr>
<tr>
<td>3</td>
<td>\times</td>
<td>b</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(2)</td>
</tr>
<tr>
<td>4</td>
<td>+</td>
<td>(1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(3)</td>
</tr>
<tr>
<td>5</td>
<td>=</td>
<td>a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(4)</td>
</tr>
</tbody>
</table>
Example of Implementation in C

```c
/* Operations supported by the three-address code */
typedef enum { MULTIPLY, ADD, MINUS, ... } Operator;

/* Definition of a parameter or a return value */
typedef union {
    unsigned long address; /* address of a variable */
    long integer_value;     /* integer constant */
    double float_value;     /* floating-point constant */
} Value;

/* Definition of a single triple */
typedef struct {
    Operator operator;
    Value arg1;
    Value arg2;
} Triple;
```
A benefit of quadruples over triples can be seen in an optimizing compiler, where instructions are often moved around.

With quadruples, if we move an instruction that computes a temporary \( t \), then the instructions that use \( t \) require no change.

With triples, the result of an operation is referred to by its position, so moving an instruction may require us to change all references to that result.

This problem does not occur with indirect triples.
Indirect tripl es consist of a listing of points to triples, rather than a listing of triples themselves.

With indirect triples, an optimizing compiler can move an instruction by reordering the instruction list, without affecting the triples themselves.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>35</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

When implemented in Java, an array of instruction objects is analogous to an indirect triple representation, since Java treats the array elements as references to objects.
Outline

1. Introduction

2. Translation scheme

3. Syntax tree and graph

4. Three-address code
   - Language basics
   - Quadruple form
   - Triple form
   - Static single-assignment form

5. Code generation of variables
Definition

Static single-assignment form (SSA) is an intermediate representation that facilitates certain code optimizations.

Two distinctive aspects distinguish SSA from the standard form of the three-address code.

1. All assignments in SSA are to variables with distinct names.
2. Introduction of the \( \phi \)-function.
All assignments in SSA are to variables with distinct names.

\begin{align*}
\text{Standard Form} & \quad \text{SSA Form} \\
\text{p} = a + b & \quad p_1 = a + b \\
\text{q} = p - c & \quad q_1 = p_1 - c \\
\text{p} = q \ast d & \quad p_2 = q_1 \ast d \\
\text{p} = e - p & \quad p_3 = e - p_2 \\
\text{q} = p + q & \quad q_2 = p_3 + q_1
\end{align*}
A notational convention, called \( \phi \)-function, is introduced to combine two definitions of the same variable in parallel control-flow paths.

For example, the source program:

```plaintext
if flag then x = -1;
else x = 1;
y = x * a;
```

has two control-flow paths in which the variable \( x \) is defined. It is impossible to known which of the two \( x \) is used in \( x \ast a \).

We introduce the \( \phi \)-function that replies the “defined” value:

```plaintext
if flag then \( x_1 = -1; \)
else \( x_2 = 1; \)
y = \( \phi(x_1, x_2) \ast a; \)
```
Outline

1. Introduction
2. Translation scheme
3. Syntax tree and graph
4. Three-address code
5. Code generation of variables
   - Types and declarations
   - Expressions
   - Type checking
6. Code generation of statements
Outline

3 Syntax tree and graph

4 Three-address code

5 Code generation of variables
   - Types and declarations
   - Expressions
   - Type checking

6 Code generation of statements
Types and Declarations

The application of types can be grouped as follows:

1. **Type checking:** It uses logical rules to reason about the behavior of a program at runtime. Specifically, it ensures that the types of the operands match the type expected by an operator.

2. **Translation applications:** From the type of a name, a compiler can determine the storage that will be needed for that name at runtime. Type information is also needed to calculate the address denoted by an array reference, to insert explicit type conversions, and to choose the right version of an arithmetic operator...
Outline

3 Syntax tree and graph

4 Three-address code

5 Code generation of variables
   - Types and declarations
   - Expressions
   - Type checking

6 Code generation of statements
Types have structure represented by the type expressions.

A type expression is either a basic type or is formed by applying an operator called a type constructor to a type expression.

The set of basic types and constructors depend on the language to be checked.
What is a Type Expression?

1. **Basic type**: `boolean, char, integer, float, void.`
2. **Type name**.
3. **Expression built with the array type constructor**.
4. **Record**: data structure with named fields.
5. **Function prototype**: by using the function prototype constructor `inputType → outputType`.
6. **Cartesian product** of two type expressions: if `s` and `t` are type expressions, then `s × t` is a type expression.
3 Syntax tree and graph

4 Three-address code

5 Code generation of variables
   - Types and declarations
   - Expressions
   - Type checking

6 Code generation of statements
We must define how to convert a value from one type to others. Many type-checking rules have the form, “if two type expressions are equal then return a certain type else error.”

**Definition (Type Equivalence)**

Two types are structurally equivalent when:

1. They are the same basic type.
2. They are formed by applying the same constructor to structurally equivalent types.
3. One is a type name that denotes the other.

- Points 1 and 2 are used to defined the equivalence between two type names, ie. the name equivalence.
Outline

3 Syntax tree and graph

4 Three-address code

5 Code generation of variables
   - Types and declarations
   - Expressions
   - Type checking

6 Code generation of statements
**Declarations**

- Declaration of types is handled by a grammar like:

  \[
  D \rightarrow T \text{id} ; D \\
  | \epsilon \\
  T \rightarrow B C \\
  | \text{record} \{ D \} \\
  B \rightarrow \text{int} | \text{float} \\
  C \rightarrow [ \text{num} ] C \\
  | \epsilon
  \]

- This grammar supports basic types, arrays and records.
  - `float`;
  - `int [3][4];`;
  - `record \{ float name1; record \{ int name2; \} \}`
Outline

3 Syntax tree and graph

4 Three-address code

5 Code generation of variables
   - Types and declarations
   - Expressions
   - Type checking

6 Code generation of statements
From the type of a name, we can determine the amount of storage that will be needed for the name at runtime.

At compile time, we can use these amounts to assign each name a relative address.

Both the type and relative address are saved in the symbol table.

Data of varying length (string, dynamic array...) is handled by reserving a known fixed amount of storage for a pointer to the data.

Runtime storage management is not discussed in this chapter, but in the chapter 548.
Data of varying length (string, dynamic array...) is handled by reserving a known fixed amount of storage for a pointer to the data.

The width of a type (and not of a variable) is the number of storage units (usually bytes) needed for objects of that type.

A basic type requires an integral number of storage units.

For easy access, storage for aggregates such as arrays and classes is allocated in one contiguous block of storage units.
The storage layout for data objects is strongly influenced by the addressing constraints of the target machine.

Examples

- Instructions to add integers may expect integers to be aligned, i.e. placed at certain positions in memory such as an address divisible by 4.

- An array of ten characters needs only enough bytes to hold ten characters, a compiler may therefore allocate 12 bytes (the next multiple of 4).

- Space left unused due to alignment considerations is referred to as padding.

- A compiler may pack data so that no padding is left; additional instructions may then need to be executed at runtime to position packed data so that it can be operated on as if it were properly assigned.
The SDT below computes types and their widths for basic and array types. Records will be discussed later.

\[
\begin{array}{|c|c|}
\hline
T & \rightarrow & B \\
& | & C \\
\hline
B & \rightarrow & \text{int} \\
& | & \text{float} \\
\hline
C & \rightarrow & [ \text{num} ] C \\
& | & \epsilon \\
\hline
\end{array}
\]

- The SDT uses synthesized attributes type and width for each nonterminal and two variable \( t \) and \( w \) to pass type and width information from a \( B \) node in a parse tree to the node for the production \( C \rightarrow \epsilon \).

- In a syntax-directed definition, \( t \) and \( w \) would be inherited attributes for \( C \).
3 Syntax tree and graph

4 Three-address code

5 Code generation of variables
   - Types and declarations
   - Expressions
   - Type checking

6 Code generation of statements
Modern languages allow all the declarations in a single procedure to be processed as a group.

The declarations may be distributed within a procedure eg., in Java, but they can still be processed when the procedure is analyzed.

We use a variable, say offset, to keep track of the next available relative address.
The following SDT illustrates the use of `offset`:

<table>
<thead>
<tr>
<th>$P$</th>
<th>$D$</th>
<th>(\text{offset} = 0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D$</td>
<td>$T \ \text{id}$</td>
<td>(s = \text{new} \ \text{Symbol(id.lexeme)})</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(s.\text{offset} = \text{offset} ; s.\text{type} = T.\text{type})</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(\text{SymbolTable.getCurrent().declare(id.lexeme, s)})</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(\text{offset} = \text{offset} + T.\text{width};)</td>
</tr>
</tbody>
</table>

- The semantic action associated to the head $D$ creates a symbol table entry.
- The symbol table takes the name of the variable (its lexeme), the type of the variable, and the position of the variable in the storage.
Outline

3 Syntax tree and graph

4 Three-address code

5 Code generation of variables
   - Types and declarations
   - Expressions
   - Type checking

6 Code generation of statements
The previous sequence of declarations may be used to define the fields in records and classes.

We may extend the previous grammar with the \( T \)-production.

\[
T \rightarrow \begin{cases} B & \text{t = B.type, w = B.width} \\ C & T.type = C.type; T.width = C.width \\ \text{record} \{ T \} & \text{B.type = integer; B.width = 4} \\ & \text{B.type = float; B.width = 8} \\ & \text{head.type = array (num.value,C.type)} \\ & \text{head.width = num.value * C.width} \\ & \text{head.type = t; head.width = w} \\ \epsilon & \end{cases}
\]

The field names in a record must be distinct.

The offset or relative address for a field name is relative to the data area for that record.
For convenience, the record is defined with a specific symbol table, or environment.

\[
T \rightarrow \text{record} \{ \\
\text{SymbolTable.getCurrent().offset = offset;} \\
\text{SymbolTable.openConnection();} \\
\text{offset = 0;} \\
T.\text{type = record (SymbolTable.getCurrent());} \\
T.\text{width = offset;} \\
\text{SymbolTable.closeContext();} \\
\text{offset = SymbolTable.getCurrent().offset;} \\
\}
\]

The SDT may be updated as above, where SymbolTable is defined in chapter 12.

Classes are stored as records, since no storage is reserved for methods.
Outline

3 Syntax tree and graph

4 Three-address code

5 Code generation of variables
   - Types and declarations
   - Expressions
   - Type checking

6 Code generation of statements
This section describes how the expressions are translated into three-address code.

An expression with more than one operator, like \(a + b * c\), will translate into instructions with at most one operator per instruction.

An array reference \(A[i][j]\) will expand into a sequence of three-address instructions that calculate an address for the reference.

The translation function may be placed in two locations:

1. inside the semantic actions themselves; or
2. inside a dedicated method, usually called `generate()` of the syntax tree.
Outline

3 Syntax tree and graph

4 Three-address code

5 Code generation of variables
   - Types and declarations
   - Expressions
   - Type checking

6 Code generation of statements
Translation of the Expressions

- Each operation in the expression are translated to its equivalent three-address code.
- The SDT below builds up the three-address code for the assignment and several arithmetic operations.

\[
\begin{array}{ll}
  S & \rightarrow \quad \text{id} = E ; \\
  E & \rightarrow \quad E + E \\
  E & \rightarrow \quad - E \\
  | & \quad ( E ) \\
  | & \quad \text{id}
\end{array}
\]

```
head.code = E.code || quadruple('=', E.addr, ∅, SymbolTable.getCurrent().get(id.lexeme))
head.addr = new TemporaryVariable();
head.code = E1.code || E2.code || quadruple('+', E1.addr, E2.addr, head.addr)
head.addr = new TemporaryVariable();
head.code = E.code || quadruple('minus', E.addr, ∅, head.addr)
head.addr = E.addr; head.code = E.code
head.addr = SymbolTable.getCurrent().get(id.lexeme)
head.code = "",
```
The operator \( \parallel \) denotes the concatenation of strings of characters.

\[
\begin{align*}
S & \rightarrow \text{id} = E; \\
E & \rightarrow E + E \\
E & \rightarrow - E \\
| & \quad ( E ) \\
| & \quad \text{id}
\end{align*}
\]

|         | head.code = E.code \( \parallel \) quadruple(‘\(=\)’, E.addr, \(\emptyset\), SymbolTable.getCurrent().get(id.lexeme)) | head.addr = new TemporaryVariable(); head.code = E.code \(\parallel\) quadruple(‘\(+\)’, E1.addr, E2.addr, head.addr) | head.addr = new TemporaryVariable(); head.code = E.code \(\parallel\) quadruple(‘\(\text{minus}\)’, E.addr, \(\emptyset\), head.addr) | head.addr = E.addr; head.code = E.code | head.addr = SymbolTable.getCurrent().get(id.lexeme); head.code = ”
Generating Three-Address Code

- quadruple(op, arg₁, arg₂, result) generates a quadruple form of three-address code.

| $S$     | $\rightarrow$ | $id = E$ ; | $head.code = E.code \parallel quadruple('=', E.addr, $\emptyset$, SymbolTable.getCurrent().get(id.lexeme))$
| $E$     | $\rightarrow$ | $E + E$   | $head.addr = new TemporaryVariable()$
| $E$     | $\rightarrow$ | $- E$     | $head.code = E.code \parallel quadruple('-', E.addr, $\emptyset$, head.addr)$
|         | $\mid$        | $(E)$     | $head.addr = E.addr; head.code = E.code$
|         | $\mid$        | $id$      | $head.addr = SymbolTable.getCurrent().get(id.lexeme)$
|         |              |           | $head.code = "$"$
The attribute **code** represents the generated three-address code for each nonterminal.

| $S$  | $\rightarrow$ | id = $E$ ; | head.code = $E$.code || quadruple('=', $E$.addr, ∅, SymbolTable.getCurrent().get(id.lexeme)) |
|------|---------------|-------------|------------------------------------------------------------------|
| $E$  | $\rightarrow$ | $E + E$     | head.addr = **new** TemporaryVariable(); head.code = $E_1$.code || $E_2$.code || quadruple('+', $E_1$.addr, $E_2$.addr, head.addr) |
| $E$  | $\rightarrow$ | - $E$       | head.addr = **new** TemporaryVariable(); head.code = $E$.code || quadruple('minus', $E$.addr, ∅, head.addr) |
|      |               | ( $E$ )     | head.addr = $E$.addr; head.code = $E$.code |
|      |               | id          | head.addr = SymbolTable.getCurrent().get(id.lexeme) |
|      |               |             | head.code = "" |
Because the three-address code must use temporary variables, we must create these variables by invoking the constructor of the class TemporaryVariable.

\[ S \rightarrow \text{id} = E ; \]
\[ E \rightarrow E + E \]
\[ E \rightarrow - E \]
\[ \mid ( E ) \]
\[ \mid \text{id} \]

```
S \rightarrow \text{id} = E ;
E \rightarrow E + E
E \rightarrow - E
\mid ( E )
\mid \text{id}

head.code = E.code || quadruple('=', E.addr, ∅, SymbolTable.getCurrent().get(id.lexeme))
head.addr = new TemporaryVariable();
head.code = E.code || E_1.code || E_2.code ||
quadruple('+', E_1.addr, E_2.addr, head.addr)
head.addr = new TemporaryVariable();
head.code = E.code ||
quadruple('-', E.addr, ∅, head.addr)
head.addr = SymbolTable.getCurrent().get(id.lexeme)
head.code = ""
```
The attribute *address* denotes the address that will hold the value of the expression associated to the symbol.
Example of Translation of an Expression

Input: \( a = b + - c \)

\[
\begin{align*}
S & \rightarrow \text{id} = E ; \\
E & \rightarrow E + E \\
E & \rightarrow - E \\
| & ( E ) \\
| & \text{id}
\end{align*}
\]

| head.code = E.code || quadruple(‘=’, E.addr, \\
| \( 0 \), SymbolTable.getCurrent().get(id.lexeme)) \\
| head.addr = new TemporaryVariable(); \\
| head.code = E1.code || E2.code || \\
| quadruple(‘+’, E1.addr, E2.addr, head.addr) \\
| head.addr = new TemporaryVariable(); \\
| head.code = E.code || \\
| quadruple(‘-’, E.addr, \( 0 \), head.addr) \\
| head.addr = E.addr; head.code = E.code \\
| head.addr = SymbolTable.getCurrent().get(id.lexeme) \\
| head.code = "

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
Example of Translation of an Expression

\[ S \rightarrow \text{id} = E ; \]
\[ E \rightarrow E + E \]
\[ E \rightarrow - E \]
\[ | \quad ( E ) \]
\[ | \quad \text{id} \]

Head code:
- \( S \rightarrow \text{id} = E ; \)
- \( E \rightarrow E + E \)
- \( E \rightarrow - E \)
- \( | \quad ( E ) \)
- \( | \quad \text{id} \)

[Example code]
```
head.code = E.code || quadruple('=', E.addr, ∅, SymbolTable.getCurrent().get(id.lexeme))
head.addr = new TemporaryVariable();
head.code = E_1.code || E_2.code || quadruple('+', E_1.addr, E_2.addr, head.addr)
head.addr = new TemporaryVariable();
head.code = E.code || quadruple('minus', E.addr, ∅, head.addr)
head.addr = E.addr; head.code = E.code
head.addr = SymbolTable.getCurrent().get(id.lexeme)
head.code = ''
```

Input: \( a = b + - c \)
Example of Translation of an Expression

Input: \( a = b + - c \)

\[
S \rightarrow \text{id} = E ;
\]

\[
E \rightarrow E + E
\]

\[
E \rightarrow - E
\]

| \( E \) |
| \( \text{id} \) |

head.code = E.code || quadruple(‘=’, E.addr, \( \emptyset \), SymbolTable.getCurrent().get(id.lexeme))
head.addr = \text{new} TemporaryVariable();
head.code = E1.code || E2.code || quadruple(‘+’, E1.addr, E2.addr, head.addr)
head.addr = \text{new} TemporaryVariable();
head.code = E.code || quadruple(‘minus’, E.addr, \( \emptyset \), head.addr)
head.addr = E.addr; head.code = E.code
head.addr = SymbolTable.getCurrent().get(id.lexeme)
head.code = "

addr=b
code=""
Example of Translation of an Expression

\[ S \rightarrow \text{id} = E ; \]
\[ E \rightarrow E + E \]
\[ E \rightarrow - E \]
\[ \text{id} \quad \text{or} \quad (E) \quad \text{id} \]

**Input:** \( a = b + - c \)

**Translation of Expression:**

\[ S \rightarrow \text{id} = E ; \]
\[ E \rightarrow E + E \]
\[ E \rightarrow - E \]
\[ \text{id} \quad \text{or} \quad (E) \quad \text{id} \]

**Translation Scheme:**

1. **S**: \( \text{id} = E ; \)
2. **E**: \( E + E \)
3. **E**: \( - E \)
4. **id**
5. **(E)**

**Quadruple Generation:**

- \( \text{head.code} = E\cdot\text{code} \parallel \text{quadruple('=', E.addr, } \emptyset, \text{SymbolTable.getCurrent().get(\text{id}.lexeme)} \)\)
- \( \text{head.addr} = \text{new TemporaryVariable()}; \)
- \( \text{head.code} = E_1\cdot\text{code} \parallel E_2\cdot\text{code} \parallel \text{quadruple('+', E_1.addr, E_2.addr, head.addr)} \)
- \( \text{head.addr} = \text{new TemporaryVariable()}; \)
- \( \text{head.code} = E\cdot\text{code} \parallel \text{quadruple('-', E.addr, } \emptyset, \text{head.addr)} \)
- \( \text{head.addr} = E\cdot\text{addr}; \text{head.code} = E\cdot\text{code} \)
- \( \text{head.addr} = \text{SymbolTable.getCurrent().get(\text{id}.lexeme)} \)
- \( \text{head.code} = "" \)

© 2012–2014 Stéphane Galland

stephane.galland@utbm.fr – http://www.multiagent.fr
### Example of Translation of an Expression

**Input:** \( a = b + - c \)

#### Syntax Tree

- **Expression:** \( S \rightarrow id = E; \)
- **Expression:** \( E \rightarrow E + E \)
- **Expression:** \( E \rightarrow - E \)
- **Expression:** \( | (E) \)
- **Expression:** \( | id \)

#### Code Generation

- **Translation Rule:** \( S \rightarrow id = E; \)
  - **Head Code:** \( head.code = E.code \parallel quadruple(’=', E.addr, 0, SymbolTable.getCurrent().get(id.lexeme)) \)
  - **Head Address:** \( head.addr = \text{new TemporaryVariable}() \)

- **Translation Rule:** \( E \rightarrow E + E \)
  - **Head Code:** \( head.addr = \text{new TemporaryVariable}(); head.code = E1.code \parallel E2.code \parallel quadruple(’+’, E1.addr, E2.addr, head.addr) \)

- **Translation Rule:** \( E \rightarrow - E \)
  - **Head Code:** \( head.addr = \text{new TemporaryVariable}(); head.code = E.code \parallel quadruple(’minus’, E.addr, 0, head.addr) \)

- **Translation Rule:** \( | (E) \)
  - **Head Code:** \( head.addr = E.addr; head.code = E.code \)

- **Translation Rule:** \( | id \)
  - **Head Code:** \( head.addr = SymbolTable.getCurrent().get(id.lexeme); head.code = "" \)
  - **Address:** \( addr=b \)
  - **Code:** \( code="" \)
  - **Expression:** \( id \rightarrow - E \)
  - **Address:** \( addr=c \)
  - **Code:** \( code="t1 = minus c" \)
  - **Expression:** \( E \rightarrow E + E \)
  - **Address:** \( addr=c \)
  - **Code:** \( code="" \)
Example of Translation of an Expression

Input: \( a = b + - c \)

\[
\begin{align*}
S & \rightarrow \text{id} = E ; \\
E & \rightarrow E + E \\
E & \rightarrow - E \\
| & \rightarrow ( E ) \\
| & \rightarrow \text{id}
\end{align*}
\]

| head.code = E.code \| quadruple(‘=’, E.addr, \emptyset, SymbolTable.getCurrent().get(id.lexeme)) |
| head.addr = \text{new} TemporaryVariable() |
| head.code = E1.code \| E2.code \| quadruple(‘+’, E1.addr, E2.addr, head.addr) |
| head.addr = \text{new} TemporaryVariable() |
| head.code = E.code \| quadruple(‘minus’, E.addr, \emptyset, head.addr) |
| head.addr = E.addr; head.code = E.code |
| head.addr = SymbolTable.getCurrent().get(id.lexeme) |
| head.code = “” |
Example of Translation of an Expression

<table>
<thead>
<tr>
<th>Expression</th>
<th>Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S \rightarrow \text{id} = E;$</td>
<td>head.code = E.code</td>
</tr>
<tr>
<td>$E \rightarrow E + E$</td>
<td>head.addr = new TemporaryVariable(); head.code = E1.code</td>
</tr>
<tr>
<td>$E \rightarrow - E$</td>
<td>head.addr = new TemporaryVariable(); head.code = E.code</td>
</tr>
<tr>
<td>$</td>
<td>(E)</td>
</tr>
<tr>
<td>$</td>
<td>\text{id}</td>
</tr>
</tbody>
</table>

Input: $a = b + - c$

Code:
```
t1 = minus c; t2 = b + t1;
a = t2
```

Syntax Tree:
Outline

3 Syntax tree and graph

4 Three-address code

5 Code generation of variables
   ■ Types and declarations
   ■ Expressions
   ■ Type checking

6 Code generation of statements
Code attributes can be long string, so they are usually generated incrementally.

Instead of building up $E.code$ as previously, we can modify quadruple to output the new three-address instructions in a external data structure.

<table>
<thead>
<tr>
<th>$S$</th>
<th>$id = E$ ;</th>
<th>quadruple('=', E.addr, $\emptyset$, SymbolTable.getCurrent().get(id.lexeme))</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E$</td>
<td>$E + E$</td>
<td>head.addr = new TemporaryVariable(); quadruple('+', E_1.addr, E_2.addr, head.addr)</td>
</tr>
<tr>
<td>$E$</td>
<td>$- E$</td>
<td>head.addr = new TemporaryVariable(); quadruple('minus', E.addr, $\emptyset$, head.addr)</td>
</tr>
<tr>
<td></td>
<td>$\mid (E)$</td>
<td>head.addr = E.addr</td>
</tr>
<tr>
<td></td>
<td>$\mid id$</td>
<td>head.addr = SymbolTable.getCurrent().get(id.lexeme)</td>
</tr>
</tbody>
</table>
3 Syntax tree and graph

4 Three-address code

5 Code generation of variables
   ■ Types and declarations
   ■ Expressions
   ■ Type checking

6 Code generation of statements
Array elements can be accessed quickly if they are stored in a block of consecutive locations.

If the width of each array element is $w$, then the $i^{th}$ element of array $A$ begins in location:

$$base + (i - 1) \times w$$

where $base$ is the relative address of the storage allocated for the array.

In $k$ dimensions, let $s_j$ the number of cells at the $j^{th}$ dimension.

The formula is:

$$base + w \times \sum_{1 \leq j \leq k} \left[ (i_j - 1) \prod_{j \leq o < k} w_o \right]$$
The major problem in generating code for array references is to relate the address-calculation formulas to a grammar for array references.

- Let nonterminal $L$ generate an array name followed by a sequence of index expressions

$$L \rightarrow L [ E ] \mid \text{id} [ E ]$$

- Assume that the lowest-numbered array element is 0.
Translation for 1-Dimension Array

A one-dimension-array reference is translated as follows:

\[
L \rightarrow \text{id} \ [ \ E \ ]
\]

| head.base = SymbolTable.getCurrent().
get(id.lexeme)  
head.type = head.base.elementType  
head.addr = new TemporaryVariable()  
quadruple('\*', E.addr, head.type.width, head.addr) |

- **Attribute ”base”**: the symbol of the array.
- **Attribute ”type”**: the type of the elements of the array (given by the symbol table entry).
- **Attribute ”addr”**: the address of the element in the storage from the beginning of the array.
A *n*-dimension-array reference is translated as follows:

\[
L \rightarrow L[E]
\]

<table>
<thead>
<tr>
<th>head.base = L.base</th>
</tr>
</thead>
<tbody>
<tr>
<td>head.type = L.type.elementType</td>
</tr>
<tr>
<td>t = new TemporaryVariable()</td>
</tr>
<tr>
<td>head.addr = new TemporaryVariable()</td>
</tr>
<tr>
<td>quadruple('*', E.addr, head.type.width, t)</td>
</tr>
<tr>
<td>quadruple('+', L.addr, t, head.addr)</td>
</tr>
</tbody>
</table>

- **Attribute "base":** the symbol of the array.
- **Attribute "type":** the type of the elements of the array (given by the symbol table entry).
- **Attribute "addr":** the address of the element in the storage from the beginning of the array.
The array-reference productions are referred as:

\[
\begin{align*}
S & \rightarrow L = E; \\
E & \rightarrow L
\end{align*}
\]

 quadruple('[]=', L.addr, E.addr, L.base)
 head.addr = new TemporaryVariable()
 quadruple('[]=]', L.base, L.addr, head.addr)

- **Attribute ”base”**: the symbol of the array.
- **Attribute ”addr”**: the address of the element in the storage from the beginning of the array; or the address of a temporary variable.
Example of Translation of an Expression

Input: $c + a[i][j]$
Example of Translation of an Expression

Input: \( c + a[i][j] \)

Output:
Example of Translation of an Expression

Input: \( c + a[i][j] \)

Output:

\[
E \rightarrow id \\
\text{head.addr} = \text{SymbolTable.getCurrent().get(id.lexeme)}
\]
Example of Translation of an Expression

Input: $c + a[i][j]$

Output:

$E \rightarrow id$  
head.addr = SymbolTable.getCurrent().get(id.lexeme)
Example of Translation of an Expression

Input: $c + a[i][j]$

Output:

$t_1 = i \times 12$

Assume that:

a) $a$ was declared as int[2][3]

b) An integer takes 4 bytes
Example of Translation of an Expression

Input: $c + a[i][j]$
Example of Translation of an Expression

Input: \( c + a[i][j] \)

Output:

\[
\begin{align*}
t_1 &= i \ast 12 \\
t_2 &= j \ast 4 \\
t_3 &= t_1 + t_2
\end{align*}
\]
Example of Translation of an Expression

Input: \( c + a[i][j] \)

Output:

\[
\begin{align*}
t_1 &= i \times 12 \\
t_2 &= j \times 4 \\
t_3 &= t_1 + t_2 \\
t_4 &= a[t_3]
\end{align*}
\]

\[
\begin{array}{c}
E \\
E \rightarrow L \\
head.addr = \textbf{new} \ \text{TemporaryVariable}() \\
quadruple(\[=\], L.base, L.addr, head.addr)
\end{array}
\]
Example of Translation of an Expression

Input: $c + a[i][j]$

```
head.addr = new TemporaryVariable();
quadruple('+', $E_1.addr$, $E_2.addr$, head.addr)
```

Output:

\[
t_1 = i \times 12 \\
t_2 = j \times 4 \\
t_3 = t_1 + t_2 \\
t_4 = a[t_3] \\
t_5 = c + t_4
\]
Outline

3 Syntax tree and graph

4 Three-address code

5 Code generation of variables
   - Types and declarations
   - Expressions
   - Type checking
     - Introduction
     - Forms of Type Checking
     - Type conversion
     - Overloaded and polymorphic functions

6 Code generation of statements
The compiler must determine if the types of the variables are consistent according to a collection of logical rules that is called the type system.

- To do type checking a compiler needs to assign a type expression to each component of the source program.
- Type checking enables to catch errors in programs.
- Type checking has two forms: synthesis and inference.

**Note:** In this section, we consider only the type checking of expressions. The rules for checking statements are similar.
Dynamic Type Checking: Any check can be done dynamically, if the target code carries the type of an element along with the value of the element.

Static Type Checking: A sound type system eliminates the need for dynamic checking for type errors. It allows us to determine statically that these errors cannot occur when the target program runs.

Strongly Typed Language: An implementation of a language is strongly typed if a compiler guarantees that the programs will run without type errors.

We recommend to use a sound type system, when possible.
3 Syntax tree and graph

4 Three-address code

5 Code generation of variables
   - Types and declarations
   - Expressions
   - Type checking
     - Introduction
     - Forms of Type Checking
     - Type conversion
     - Overloaded and polymorphic functions

6 Code generation of statements
Definition

Type synthesis builds up the type of an expression from the types of its subexpressions.

- It requires names, and their types, to be declared before they are used.

Example

- The type of $E_1 + E_2$ is defined according to the types of $E_1$ and $E_2$.
  - $E_1$ is int and $E_2$ is float then $E_1 + E_2$ is float.
Definition

Type inference determines the type of a language construct from the way it is used.

- Type inference is needed for languages like ML, which check types, but do not require names to be declared.

Example

- Let the PHP code:  
  ```php
  \$
  a = 14;
  \$
  b = "a" . \$
  a;
  \$
  c = 1 + \$
  a;
  ```

- $a$ is used as a string for $b$’s expression, and used as an integer for $c$’s expression.
Outline

3 Syntax tree and graph

4 Three-address code

5 Code generation of variables
   - Types and declarations
   - Expressions
   - Type checking
     - Introduction
     - Forms of Type Checking
     - Type conversion
     - Overloaded and polymorphic functions

6 Code generation of statements

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
Why Type Conversion?

- Consider expressions like \( x + i \), where \( x \) is of type \texttt{float} and \( i \) is of type \texttt{integer}.

- Since the representations of floating-point numbers and integers are different, and different machine instructions are used for operations on integers and floats, the compiler may need to convert one of the operands to ensure that both operands are of the same type when the operator is applied.

\[
\begin{align*}
  t_1 &= \text{(float) 2} \\
  t_2 &= t_1 \times 3.14
\end{align*}
\]

- **Note:** Type conversion rules vary from language to language.
- **Widening conversion**: preserve information between the value before the conversion and the value after the conversion.
- **Narrowing conversion**: can lose information.
Implicit and Explicit Type Conversions

- **Implicit conversion**: conversions automatically done by the compiler, with a possible warning message in the case of narrowing conversion.
  - Implicit conversion is also called *coercion*.
  - Many languages limit the implicit conversions to *widening conversions*.

- **Explicit conversion**: conversions written in the source code by the programmer.
  - Explicit conversion is also called *cast*.
**Function** \( \text{maxType}(t_1, t_2) \)

**Definition**

\[ \text{maxType} : T \times T \rightarrow T \]

\((t_1, t_2) \mapsto \text{maximum or least upper bounds of } t_1 \text{ and } t_2 \]

in the widening hierarchy. Otherwise error.

**Example**

\[ \text{maxType}(\text{short, char}) \rightarrow \text{int} \]
Function \textbf{widenVar}(a, t_{out}, t_{in})

\textbf{Definition}

\textit{widenVar} : A \times T \times T \rightarrow A

(a, t_{out}, t_{in}) \mapsto \text{generates the code that widens the value pointed by } a \text{ to the type } t_{out},\text{ assuming that } a \text{ is of type } t_{in}. \text{ This conversion is done only if it is required. Returns the address were the result of the is available.}
Assume a language with only the two types \texttt{int} and \texttt{float}.

\begin{verbatim}
Function widenVar(a : \mathbb{A}, t_{out} : \mathbb{T}, t_{in} : \mathbb{T}) : \mathbb{A}
begin
  if \( t_{out} = t_{in} \) then
    return a;
  else if \( t_{in} = \texttt{int} \) and \( t_{out} = \texttt{float} \) then
    t ← new TemporaryVariable();
    quadruple(''(float)'', a, \emptyset, t);
    return t;
  else
    Throw "Cannot widen the variable";
end
\end{verbatim}
Function narrowVar(a, t_{out}, t_{in})

Definition

\[ \text{narrowVar} : \mathbb{A} \times \mathbb{T} \times \mathbb{T} \rightarrow \mathbb{A} \]

This function generates the code that narrows the value pointed by \( a \) to the type \( t_{out} \), assuming that \( a \) is of type \( t_{in} \). This conversion is done only if it is required. Returns the address where the result is available.

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
Assume a language with only the two types \texttt{int} and \texttt{float}.

\textbf{Function} narrowVar(a : $\mathbb{A}$, $t_{out}$ : $\mathbb{T}$, $t_{in}$ : $\mathbb{T}$) : $\mathbb{A}$

\begin{verbatim}
begin
  if $t_{out} \neq \texttt{int}$ or $t_{in} \neq \texttt{float}$ then
    Throw "Cannot narrow the variable";
  else if $t_{out} = \texttt{int}$ and $t_{in} = \texttt{float}$ then
    t ← new TemporaryVariable();
    quadruple('(float)', a, $\emptyset$, t);
    return t;
  else
    return a;
end
\end{verbatim}
The attribute "type" is added to store the type of an expression. The SDD is updated to check the types:

\[
\begin{array}{c|c|c}
E & \rightarrow & E + E \\
\hline
E & \rightarrow & id = E \\
\end{array}
\]

\[
\begin{align*}
\text{head.type} &= \text{maxType}(E_1\text{.type}, E_2\text{.type}) \\
o_1 &= \text{widenVar}(E_1\text{.addr}, E_1\text{.type}, \text{head.type}) \\
o_2 &= \text{widenVar}(E_2\text{.addr}, E_2\text{.type}, \text{head.type}) \\
\text{head.addr} &= \text{new} \ \text{TemporaryVariable()} \\
\text{quadruple}(\ '+', o_1, o_2, \text{head.addr}) \\
\text{head.addr} &= \text{SymbolTable.getCurrent().get}(id\text{.lexeme}) \\
\text{head.type} &= v\text{.type} \\
w &= \text{narrowVar}(E\text{.addr}, E\text{.type}, \text{head.type}) \\
\text{if } (w \neq E\text{.addr}) \\
& \quad \text{warning } "\text{May lose information}" \\
\text{else} \\
& \quad w = \text{widenVar}(E\text{.addr}, E\text{.type}, \text{head.type}) \\
\text{quadruple}(\ '=' , w, \emptyset, \text{head.addr})
\end{align*}
\]
Outline

3 Syntax tree and graph

4 Three-address code

5 Code generation of variables
   - Types and declarations
   - Expressions
   - Type checking
     - Introduction
     - Forms of Type Checking
     - Type conversion
     - Overloaded and polymorphic functions

6 Code generation of statements
Definition

Allows creating several functions with the same name, which differ from each other in the type of the input and the output of the function.

- Depending on the context, the overloading may be for a function, a procedure, a method, or an operator.
- The symbol table must contains all the signatures of the functions (in the context, which is using the symbol table).
- A signature consists of:
  1. the function name,
  2. the list of the types of the arguments of the function.
Definition

The term “polymorphic” refers to any code fragment that can be executed with arguments of different types.

- In this section, we consider the parametric polymorphism, where the polymorphism is characterized by parameters or type variables.
Example of Polymorphic Function

Consider the following definition in ML language:

```
fun length(x) = if null(x) then 0 else length(tl(x)) + 1;
```

Consider the following statement in ML language:

```
length(["sun", "mon", "tue"] ) + length([10, 9, 8, 7] )
```

The same function length() is invoked on an array of strings and on an array of integers.

The result of the ML statement is: 3 + 4 = 7
Using the symbol $\forall$ and the type constructor `list`, the type of the function `length` can be written as:

$$\forall a. \text{list}(a) \rightarrow \text{int}$$

- The $\forall$ symbol is the universal quantifier, and the type variable to which it is applied is said to be bound by it.
- A type expression with a $\forall$ symbol in it will be referred as a “polymorphic type.”
- Each time a polymorphic function is applied, its bound type variables (a...) can denote a different type.
How to determine the types in the signature of a polymorphic function?

We must infer the types by exploring the syntax tree of the function and applying the substitution and unification operations.

- **Substitution**: A mapping from type variables to type expressions. **Example**: `list (int)` is an instance of `list (α)`, since it is the result of substituting `int` for `α` in `list (α).

- **Unification**: Determine whether type variables `s` and `t` are structurally equivalent by substituting the type variables in `s` and `t` by type expressions.
Example of Type Inference on Polymorphic Function

```latex
fun length(x) = 
  if null(x) then 0 else length(tl(x)) + 1;
```

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
<th>Unification</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
func

length

x

if

call

0

+

call

1

call

length

tl

x
```

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
Example of Type Inference on Polymorphic Function

```
fun length(x) =
  if null(x) then 0
else length(tl(x)) + 1;
```

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
<th>Unification</th>
</tr>
</thead>
<tbody>
<tr>
<td>length</td>
<td>β</td>
<td>γ</td>
</tr>
<tr>
<td>x</td>
<td>β</td>
<td></td>
</tr>
</tbody>
</table>

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
**Example of Type Inference on Polymorphic Function**

```
fun length(x) =
  if null(x) then 0
  else length(tl(x)) + 1;
```

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
<th>Unification</th>
</tr>
</thead>
<tbody>
<tr>
<td>length</td>
<td>$\beta \to \gamma$</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>$\beta$</td>
<td></td>
</tr>
<tr>
<td>if</td>
<td>$\mathbb{B} \times \alpha \times \alpha \to \alpha$</td>
<td>$\alpha = \gamma$</td>
</tr>
</tbody>
</table>
Example of Type Inference on Polymorphic Function

```
fun length(x) =
  if null(x) then 0
  else length(tl(x)) + 1;
```

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
<th>Unification</th>
</tr>
</thead>
<tbody>
<tr>
<td>length</td>
<td>(\beta \rightarrow \gamma)</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>(\beta)</td>
<td></td>
</tr>
<tr>
<td>if</td>
<td>(\mathbb{B} \times \alpha \times \alpha \rightarrow \alpha)</td>
<td>(\alpha = \gamma)</td>
</tr>
<tr>
<td>null</td>
<td>(\text{list}(\omega_n) \rightarrow \mathbb{B})</td>
<td></td>
</tr>
</tbody>
</table>
Example of Type Inference on Polymorphic Function

fun length(x) = if null(x) then 0 else length(tl(x)) + 1;

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
<th>Unification</th>
</tr>
</thead>
<tbody>
<tr>
<td>length</td>
<td>$\beta \rightarrow \gamma$</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>$\beta$</td>
<td></td>
</tr>
<tr>
<td>if</td>
<td>$B \times \alpha \times \alpha \rightarrow \alpha$</td>
<td>$\alpha = \gamma$</td>
</tr>
<tr>
<td>null</td>
<td>list($\omega_n$) $\rightarrow B$</td>
<td></td>
</tr>
<tr>
<td>null(x)</td>
<td>$B$</td>
<td>list($\omega_n$) $= \beta$</td>
</tr>
</tbody>
</table>
Example of Type Inference on Polymorphic Function

Expression | Type | Unification
---|---|---
length | $\beta \rightarrow \gamma$ | 
$x$ | $\beta$ | 
if | $B \times \alpha \times \alpha \rightarrow \alpha$ | $\alpha = \gamma$
null | list$(\omega_n) \rightarrow B$ | 
null$(x)$ | $B$ | list$(\omega_n) = \beta$
0 | int | $\alpha = \text{int}$

fun length(x) = if null(x) then 0 else length(tl(x)) + 1;

Expression Type Graph

```
func
  length x
    if
      null x
        call 0
      else
        length tl(x)
    +
      call 1
        length call
          tl x
```
Example of Type Inference on Polymorphic Function

Expression | Type | Unification
---|---|---
length | $\beta \rightarrow \gamma$ | 
$x$ | $\beta$ | 
if | $\mathbb{B} \times \alpha \times \alpha \rightarrow \alpha$ | $\alpha = \gamma$ 
null | list($\omega_n$) $\rightarrow \mathbb{B}$ | 
null(x) | $\mathbb{B}$ | list($\omega_n$) $= \beta$
0 | int | $\alpha = \text{int}$
+ | $\phi \times \phi \rightarrow \phi$ | $\phi = \alpha$

fun length(x) =
  if null(x) then 0
  else length(tl(x)) + 1;

Expression Type Unification

```
func

length x

if

call

0 +

call

1

call

x

tl

null x

length

null(x)

0

+

β

α

0

φ

α

β

γ

null

list($\omega_n$) → $\mathbb{B}$

null(x)

B

list($\omega_n$) = $\beta$

0

int

$\alpha = \text{int}$

+ $\phi \times \phi \rightarrow \phi$

$\phi = \alpha$

```
Example of Type Inference on Polymorphic Function

fun length(x) = 
  if null(x) then 0 
  else length(tl(x)) + 1;

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
<th>Unification</th>
</tr>
</thead>
<tbody>
<tr>
<td>length</td>
<td>β → γ</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>β</td>
<td></td>
</tr>
<tr>
<td>if</td>
<td>B × α × α → α</td>
<td>α = γ</td>
</tr>
<tr>
<td>null</td>
<td>list(ω_n) → B</td>
<td></td>
</tr>
<tr>
<td>null(x)</td>
<td>B</td>
<td>list(ω_n) = β</td>
</tr>
<tr>
<td>0</td>
<td>int</td>
<td>α = int</td>
</tr>
<tr>
<td>+</td>
<td>φ × φ → φ</td>
<td>φ = α</td>
</tr>
</tbody>
</table>

The type of the function "length" is:
length : list(ω_n) → int
Outline

3 Syntax tree and graph

4 Three-address code

5 Code generation of variables

6 Code generation of statements
   - Control flow
   - Backpatching

7 Conclusion
Outline

3 Syntax tree and graph

4 Three-address code

5 Code generation of variables

6 Code generation of statements
   - Control flow
   - Backpatching

7 Conclusion
Control Flow

- The translation of statements such as if-else-statements and while-statements is tied to the translation of boolean expressions.
- In programming languages, boolean expressions are often used to:
  1. Alter the flow of control.
  2. Compute logical values.
- The intended use of boolean expressions is determined by its syntactic context.
- To support this distinction, we may:
  1. Use two different nonterminals,
  2. Use inherited attributes,
  3. Use a set of flags during the parsing, or
  4. Build a syntax tree and invoke different procedures for the two different uses.
In short-circuit code, the boolean operators translate into jumps.
The operators themselves do not appear in the code.
Instead, the value of a boolean expression is represented by a position in the code sequence.

Example

```plaintext
if ( x < 100 || x > 200 && x != y ) x = 0;
```

```plaintext
if x < 100 then goto L1
ifFalse x > 200 then goto L2
ifFalse x ≠ y then goto L2
L1:x = 0
L2:...
```
Outline

3 Syntax tree and graph

4 Three-address code

5 Code generation of variables

6 Code generation of statements
   - Control flow
   - Backpatching

7 Conclusion
Statements for Control Flow

Consider the grammar:

\[
S \rightarrow \text{if ( } B \text{ ) } S \\
| \text{if ( } B \text{ ) } S \text{ else } S \\
| \text{while ( } B \text{ ) } S
\]

We introduce the attributes:

- **B.code** and **S.code**: synthesized attributes; three-address code of the nonterminals.
- **B.true**: inherited attribute; the label of the code associated to the then-statements.
- **B.false**: inherited attribute; the label of the code associated to the else-statements.
- **B.next**: inherited attribute; the label of the code just after the current if-then-else statements.
The nontermirnal for the condition is no more $E$, but $B$.

- The function `newlabel()` creates a new label each time it is called.
- The function `label(L)` attaches label L to the next three-address instruction to be generated.
Statement if-then-else

\[ S \rightarrow \text{if (} B \text{)} \]

\[ S \text{ else} \]

\[ S \]

\[ B.\text{true} = \text{newlabel()} \]
\[ B.\text{false} = \text{newlabel()} \]
\[ S_1.\text{next} = \text{head.next} \]
\[ \text{label}(B.\text{true}) \]
\[ \text{quadruple}('\text{goto}', \]
\[ \text{head.next}, \]
\[ \emptyset, \emptyset) \]
\[ S_2.\text{next} = \text{head.next} \]
\[ \text{label}(B.\text{false}) \]
Statement while

\[
S \rightarrow \text{while ( } B \text{ )}
\]

\[
\begin{align*}
\text{begin} & = \text{newlabel()} \\
\text{B.true} & = \text{newlabel()} \\
\text{B.false} & = \text{head.next} \\
\text{S.next} & = \text{begin} \\
\text{label(B.true)} & \text{ quadruple(’goto’, begin, } \emptyset, \emptyset) \\
\end{align*}
\]
Outline

3 Syntax tree and graph

4 Three-address code

5 Code generation of variables

6 Code generation of statements
   - Control flow
   - Backpatching

7 Conclusion

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
Boolean expressions dedicated to control flow need dedicated semantic rules.

Remember that the boolean expressions used in control-flow statements must be translated into jumping three-address code.

<table>
<thead>
<tr>
<th>$B$</th>
<th>$\rightarrow$</th>
<th>true</th>
<th>quadruple('goto', head.true, $\emptyset$, $\emptyset$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B$</td>
<td>$\rightarrow$</td>
<td>false</td>
<td>quadruple('goto', head.false, $\emptyset$, $\emptyset$)</td>
</tr>
</tbody>
</table>
Boolean Operator NOT

- No code is needed for an expression of the form $\neg B$.
- Just interchange the true and false attributes of the head to set the true and false attributes of $B$. 

\[
\begin{align*}
B & \rightarrow \neg B \\
B.\text{true} &= \text{head.false} \\
B.\text{false} &= \text{head.true}
\end{align*}
\]
If $B_1$ is true, the head is true.

If $B_1$ is false, evaluate $B_2$.

So $B_1\.false$ is the label of the first instruction of $B_2$.

The value of the head becomes the same as the value of $B_2$. 

\[
\begin{array}{c|c}
B \\ \rightarrow \\ B_1.true = head.true \\
B_1.false = newlabel() \\
B_2.true = head.true \\
B_2.false = head.false \\
\end{array}
\]
Boolean Operator AND

- If $B_1$ is false, the head is false.
- If $B_1$ is true, evaluate $B_2$.
- So $B_1.true$ is the label of the first instruction of $B_2$.
- The value of the head becomes the same as the value of $B_2$.

<table>
<thead>
<tr>
<th>$B$</th>
<th>$B_1.true = \text{newlabel}()$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B_1.false = \text{head.false}$</td>
<td></td>
</tr>
<tr>
<td>$B_2.true = \text{head.true}$</td>
<td></td>
</tr>
<tr>
<td>$B_2.false = \text{head.false}$</td>
<td></td>
</tr>
</tbody>
</table>

$$B \rightarrow \begin{array}{c}
\text{code for } B_1 \\
\text{to } B_1.true \\
\text{to head.false} \\
\text{...} \\
\text{code for } B_2 \\
\text{to head.true} \\
\text{to head.false}
\end{array}$$
Operators of Comparison

- The form $a < b$ translates into:
  
  $t = (a < b)$
  
  if $t$ then \texttt{goto} B.true
  
  \texttt{goto} B.false

\begin{align*}
B \rightarrow E \text{ rel } E & \quad \text{t = new TemporaryVariable()}
\text{quadruple(} \text{rel}.\text{operator,}
\ E_1.\text{addr, } E_2.\text{addr, } t) \\
\text{quadruple(}'if', \ t, \ 
\head.\text{true, } \emptyset) \\
\text{quadruple(}'goto', \ head.\text{false, } \emptyset, \ \emptyset)
\end{align*}
Example of Translation

\[
\text{if ( } x < 100 \text{ } \mid\mid \text{ } x > 200 \text{ } \&\& \text{ } x \neq y \text{ ) } x = 0;
\]

\[
t_1 = x < 100 \\
\text{if } t_1 \text{ then goto L2} \\
goto L3 \\
L3: t_1 = x > 200 \\
\text{if } x > 200 \text{ then goto L4} \\
goto L1 \\
L4: t_1 = x \neq y \\
\text{if } t_1 \text{ then goto L2} \\
goto L1 \\
L2: x = 0 \\
L1: \ldots
\]
Outline

3 Syntax tree and graph

4 Three-address code

5 Code generation of variables

6 Code generation of statements
   - Control flow
   - Backpatching

7 Conclusion
The semantic rules described in the previous slides may generate more goto instructions than strictly necessary.

Example

\[
\text{L3: } t_1 = x > 200 \\
\quad \text{if } t_1 \text{ then goto L4} \\
\quad \text{goto L1} \\
\text{L4: \ldots} \\
\text{L1: \ldots}
\]

Best Practice

\[
\text{L3: } t_1 = x > 200 \\
\quad \text{ifFalse } t_1 \text{ then goto L1} \\
\quad \ldots \\
\text{L1: \ldots}
\]
Avoiding redundant gotos is done by introducing a constant for the value of the labels: \texttt{fall}.

This constant means “don’t generate any jump.”, or “fall in the next available instruction.”

We can adapt the semantic rules of the boolean expressions.

\[ S \rightarrow \textbf{if} \ ( \ B \ ) \ S. \]

\begin{tabular}{|c|c|c|}
\hline
\multicolumn{2}{|c|}{\( S \rightarrow \textbf{if} \ ( \ B \ ) \ S \)} & \multicolumn{1}{c|}{\begin{tabular}{l}B.true = newlabel() \\
B.false = head.next \\
S.next = head.next \\
label(B.true)
\end{tabular}} \\
\hline
\end{tabular}

\begin{tabular}{|c|c|c|}
\hline
\multicolumn{2}{|c|}{\begin{tabular}{l}B.true = \texttt{fall} \\
B.false = head.next \\
S.next = head.next
\end{tabular}} & \multicolumn{1}{c|}{\begin{tabular}{l}S \rightarrow \textbf{if} \ ( \ B \ ) \ S \\
\end{tabular}} \\
\hline
\end{tabular}
Remove Redundant Gotos of the OR Operator

B →
  B
|| B

B₁.true = head.true
B₁.false = newlabel()
B₂.true = head.true
B₂.false = head.false

B →
  if head.true = fall
    B₁.true = newlabel()
  else
    B₁.true = head.true
    B₁.false = fall
    B₂.true = head.true
    B₂.false = head.false
  if head.true = fall
    label(B₁.true)
Outline

3 Syntax tree and graph

4 Three-address code

5 Code generation of variables

6 Code generation of statements
   - Control flow
   - Backpatching

7 Conclusion
A key problem is the matching of a jump instruction with the target address of the jump.

Example

- Consider the statement `if ( B ) S`.
- In a one-pass translation, `B` must be translated before `S` is examined.
- What is the address of the label that permits to go over the code for `S`?
Solution 1

- In the previous slides, we solve this problem by using inherited attributes "next".
- But a separate pass is then needed to bind labels to addresses.

Solution 2

- Backpatching can be used to generate code for boolean expressions and flow-of-control statements in one pass.
- This approach is detailed in the following slides.
General Principle of Backpatching

- When the jump target is after the current instruction, the address of the current instruction is added into a list.

- When the address of the target instruction is known, the instructions in the list are updated.
New attributes for the Backpatching

- Introduction of synthesized attributes: `bptruelist` and `bpfalselist` of the nonterminal \( B \) are used to manage labels in jumping code for boolean expressions.

  - \( B.bptruelist \): list of jump or conditional jump instructions into which we must insert the label to which control goes if \( B \) is true.

  - \( B.bpfalselist \): list of instructions that eventually get the label to which control goes when \( B \) is false.

- Similarly, the other productions, such as \( S \), must be updated in a simlar way with synthesized attributes.
Tools for the Backpatching

- **makebplist(adr)**: creates a new list containing only *adr*, an index into the array of instructions.

- **mergebplists(lst1,lst2)**: concatenates the lists pointed by *lst1* and *lst2*, and returns a pointer to the result.

- **backpatch(lst,adr)**: inserts *adr* as the target label for each of the instructions on the list pointed to by *lst*.

- **instadr()**: replies the address of the instruction that will be generated by the next call to quadruple().

- **Unknown address**: The keyword `?` represents an unknown address.
| $B \rightarrow \text{true}$ | head.bptruelist = makebplist(instadr()) 
 quadruple('goto', ?, $\emptyset$, $\emptyset$) |
|-------------------------|--------------------------------------------------|
| $B \rightarrow \text{false}$ | head.bpfalselist = makebplist(instadr()) 
 quadruple('goto', ?, $\emptyset$, $\emptyset$) |
| $B \rightarrow E \ rel \ E$ | $t = \textbf{new} \ \text{TemporaryVariable}()$ 
 quadruple(rel.operator, $E_1$.addr, $E_2$.addr, t) 
 head.bptruelist = makebplist(instadr()) 
 quadruple('if', t, ?, $\emptyset$) 
 head.bpfalselist = makebplist(instadr()) 
 quadruple('goto', ?, $\emptyset$, $\emptyset$) |
| $B \rightarrow B || B$ | backpatch(B₁.bpfalselist, instadr()) 
 head.bptruelist = mergbplists(B₁.bptruelist, B₂.bptruelist) 
 head.bpfalselist = B₂.bpfalselist |
### Backpatching of the Control-Flow Statements

<table>
<thead>
<tr>
<th>Grammar Rule</th>
<th>Backpatching Algorithm</th>
</tr>
</thead>
</table>
| \[ S \rightarrow \text{if (B) } \rightarrow S \] | `backpatch(B.bptruelist, instadr())`  
`\text{head.bpnextlist} = \text{mergebplists}(B.bpfalselist, S.bpnextlist)` |
| \[ S \rightarrow \text{if (B) } \rightarrow S \text{ else } \rightarrow S \] | `backpatch(B.bptruelist, instadr())`  
`backpatch(B.bpfalselist, instadr())`  
`\text{head.bpnextlist} = \text{mergebplists}(S_1.bpnextlist, S_2.bpnextlist)` |
| \[ S \rightarrow \text{while (B) } \rightarrow S \] | `a = instadr()`  
`backpatch(B.bptruelist, instadr())`  
`backpatch(S.bpnextlist, a)`  
`\text{quadruple('goto', a, \emptyset, \emptyset)}`  
`\text{head.bpnextlist} = B.bpfalselist` |

The attribute `bpnextlist` is the list of the addresses of the instructions that are referring the “next instruction.”
Procedure backpatch(list, address)

Input : $Q$ is the global list of the generated quadruples.

begin
  foreach $a \in \text{list}$ do
    $q \leftarrow Q[a]$;
    if $q.op = 'goto'$ then
      if $q.arg_1 \neq ?$ then throw "Cannot backpatch";
      $q.arg_1 \leftarrow \text{address}$;
    else if $q.op = 'if'$ then
      if $q.arg_2 \neq ?$ then throw "Cannot backpatch";
      $q.arg_2 \leftarrow \text{address}$;
    else if $q.op = 'ifFalse'$ then
      if $q.arg_2 \neq ?$ then throw "Cannot backpatch";
      $q.arg_2 \leftarrow \text{address}$;
    else
      throw "Instruction to backpatch not found";
  end
end
Outline

1. Introduction
2. Translation scheme
3. Syntax tree and graph
4. Three-address code
5. Code generation of variables
6. Code generation of statements
7. Conclusion
Inherited and synthesized attributes: Syntax-directed definitions may use two kinds of attributes. A synthesized attribute at a parse-tree node is computed from attributes at its children. An inherited attribute at a node is computed from attributes at its parent and/or siblings.

Dependency graphs: Given a parse tree and an SDD, we draw edges among the attribute instances associated with each parse-tree node to denote that the value of the attribute at the head of the edge is computed in terms of the value of the attribute at the tail of the edge.

S-Attributed definitions: In a S-attributed SDD, all attributes are synthesized.
Key Concepts in the Chapter (#2)

- **L-Attributed definitions:** In a L-attributed SDD, attributes may be inherited or synthesized. However, inherited attributes at a parse-tree node may depend only on inherited attributes of its parent and on (any) attributes of siblings to its left.

- **Syntax trees:** Each node in a syntax tree represents a construct; the children of the node represent the meaningful components of the construct.
Intermediate representation: An intermediate representation is typically some combination of a graphical notation and three-address code. As in syntax, a node in a graphical notation represents a construct; the children of a node represent its subconstructs. Three address code takes its name from instructions of the form $x = y \text{ op } z$, with at most one operator per instruction. There are additional instructions for control flow.
Translate expressions: Expressions with built-up operations can be unwound into a sequence of individual operations by attaching actions to each production of the form $E \rightarrow E_1 \text{ op } E_2$. The action either creates a node for $E$ with the nodes for $E_1$ and $E_2$ as children, or it generates a three-address instruction that applies op to the addresses for $E_1$ and $E_2$ and puts the result into a new temporary name, which becomes the address of $E$.

Check types: The type of an expression $E_1 \text{ op } E_1$ is determined by the operator op and the types of $E_1$ and $E_2$. A coercion is an implicit type conversion. Intermediate code contains explicit type conversions to ensure an exact match between operand types and the types expected by an operator.
Generate jumping code for boolean expression: In short-circuit or jumping code, the value of a boolean expression is implicit in the position reached in the code. Jumping code is useful because a boolean expression $B$ is typically used to $t=true$ or $t=false$, as appropriate, where $t$ is a temporary name. Using labels for jumps, a boolean expression can be translated by inheriting labels corresponding to its true and false exits attributes. The constants $true$ and $false$ translate into a jump to the true and false attributes, respectively.
Implement statements using control flow: Statements can be translated by inheriting a label next, where next marks the first instruction after the code for this statement. The conditional $S \rightarrow \text{if} (B) S$ can be translated by attaching a new label marking the beginning of the code for $S$ and passing the new label and $S.\text{next}$ for the true and false attributes, respectively, of $B$.

Alternatively, use backpatching: Backpatching is a technique for generating code for boolean expressions and statements in one pass. The idea is to maintain lists of incomplete jumps, where all the jump instructions on a list have the same target. When the target becomes known, all the instructions on its list are completed by filling in the target.
Implement records: Field names in a record or class can be treated as a sequence of declarations. A record type encodes the types and relative addresses of the fields. A symbol table object can be used for this purpose.
A general translation program for phrase structure languages.

Java intermediate bytecodes.
In *ACM SIGPLAN Workshop on Intermediate Representations*.

A syntax-directed compiler for ALGOL 60.

The intrinsic exponential complexity of the circularity problem for attribute grammars.

A tour through the portable C compiler.
Technical report, Bell Telephone Laboratories Inc., Murray Hill, N.J.

Semantics of context-free languages.

Attributed translations.
A theory of type polymorphism in programming.

Attribute grammar paradigms – a high-level methodology in language implementation.

*Types and Programming Languages.*

A tour through the portable UNIX C compiler.
Technical report, Bell Telephone Laboratories Inc., Murray Hill, N.J.

Sequential formula translation.

The problem of programming communication with changing machines: a proposed solution.
*Comm. ACM, 1*(8):12–18.

The design of a pascal compiler.
Chapter 5
Run-time Environments

Stéphane GALLAND
1 Introduction
2 Data Storage
3 Stack management
4 Heap management
5 Garbage collection
6 Conclusion
A compiler must implement the abstractions embodied in the source-language definition.

The compiler must cooperate with the operating system and other systems software to support these abstracts on the target machine.

To do so, the compiler creates and manages a run-time environment in which it assumes its target programs are being executed.

This chapter presents the key points of the run-time environment:

1. Management of the Heap,
2. Management of the Stack,
3. Dynamic Memory Deallocation.
Outline

1 Introduction

2 Data Storage

3 Stack management

4 Heap management

5 Garbage collection

6 Conclusion
The target program runs in its own **logical address space** in which each value has a location.

The organization and management of this logical address space is shared between:
- the compiler,
- the operating system,
- the target machine.

The operating system maps the logical addresses into physical addresses.

When defined, a **virtual machine** is a program that is representing the operating system and the target machine into an abstract and platform-independent machine.
In the run-time environment, the logical address space has a structure; typically:

```
0

Code

Static Data

Heap

Free Memory

Stack

n
```
In the run-time environment, the logical address space has a structure; typically:

- The compiler can place the executable target code in a statically determined area, named Code.
- This area contains the binary representations of the instructions to execute.
- The format of the code depends on the target machine: Intel binary assembler, byte code...
In the run-time environment, the logical address space has a structure; typically:

- Similarly, the size of some program data may be known at compile time.
- The area where these data are stored in named Static area, usually put just after the Code area.
- **Examples:** string literals, global constants and variables, information related to garbage collection... 
- The addresses of the static data are directly put in the code.
In the run-time environment, the logical address space has a structure; typically:

- The stack is used to store data structures called activation records.
- Activation records are generated during the procedure/function calls (explained later).
- Basically, each record contains the status of the machine: ordinal counter, machine registers, and data whose lifetimes are the same as the activation time (usually local variables).
In the run-time environment, the logical address space has a structure; typically:

- Many languages allow the programmer to allocate and deallocate data under program control (malloc, new...)
- The heap is used to manage this kind of long-lived data.
In the run-time environment, the logical address space has a structure; typically:

- The heap and the stack are growing up and consume the free memory space between them.
- When the stack cannot grow up, the classical “stack overflow” error is fired.
- When the heap cannot grow up, the classical “out of memory” error is fired.
The layout and allocation of data to memory locations in the run-time environment are key issues in storage management.

**Static Storage Allocation**

It is made by the compiler looking only at the text of the program, not at what the program does when it executes.

**Dynamic Storage Allocation**

- It is made only while the program is running.
- Many compilers use a combination of the strategies explained in the following slide for dynamic storage allocation.
Stack Storage

- Names local to a procedure are allocated space **on a stack**.
- The stack supports the normal call/return policy for procedures.

Heap Storage

- Data that may outlive the call to the procedure that created it is usually allocated **on a “heap”** of reusable storage.
- The heap is an area of virtual memory that allows data to obtain and release storage.
- **Garbage collection**: the run-time environment detects useless data in heap and releases them.
Outline

1 Introduction

2 Data Storage

3 Stack management
   - Stack allocation
   - Access to nonlocal data on the stack

4 Heap management

5 Garbage collection

6 Conclusion
1 Introduction

2 Data Storage

3 Stack management
   ■ Stack allocation
      ■ Introduction
      ■ Activation tree
      ■ Control Stack and Activation record
      ■ Calling sequence
      ■ Variable-length data on the stack
   ■ Access to nonlocal data on the stack

4 Heap management
Compilers that use procedures, functions, or methods as units manage a part of their run-time memory as a stack.

Note: procedure will be used as a generic term for procedure, function and method.

Each time a procedure is called:
  - space for its local variables is pushed into a stack.

When the procedure terminates:
  - space is popped off the stack.

The activation of procedure ≡ the call to the procedure.
Stack allocation would not be feasible if procedure calls did not nest in time.

- If an activation of procedure p calls procedure q, then that activation of q must end before the activation of p can end.

```c
int a[11];
void readArray() {
    int i; // read and fill a
}
int partition(int m, int n) {
    // let v, a[m..p−1] < v, a[p]=v, a[p+1..n] >= v
    // return p
}
void quicksort(int m, int n) {
    int i;
    if (n>m) {
        i = partition(m,n);
        quicksort(m, i−1);
        quicksort(i+1,n);
    }
}
main() {
    readArray();
    a[0] = −9999;
    a[11] = 9999;
    quicksort(1,9);
}
```
Three common cases when \( p \) calls \( q \):

1. **Normal**: The activation of \( q \) terminates normally. Then in essentially any language, control resumes just after the point of \( p \) at which the call to \( q \) was made.

2. **Abort**: The activation of \( q \), or some procedure called by \( q \), either directly or indirectly, aborts. \( p \) ends simultaneously with \( q \).
3 Exception: The activation of q terminates because of an exception that q cannot handle. Procedure p may handle the exception: the activation of q has terminated but p continues (not necessary where q was called). If p cannot handle the exception, then this activation of p terminates at the same time as the activation of q, and presumably, the exception will be handled by some other open activation of a procedure.
Introduction

2 Data Storage

3 Stack management
   - Stack allocation
     - Introduction
     - Activation tree
     - Control Stack and Activation record
     - Calling sequence
     - Variable-length data on the stack
     - Access to nonlocal data on the stack

4 Heap management
The activations of procedures during the running of an entire program is represented by a tree, named activation tree.

- **Node**: an activation;

- **Root Node**: the root is the activation of the “main” procedure that initiates the execution of the program.

- **Child Node**: activations of the procedures called by the activation represented by the parent node. The order of the children (from left to right) is the order of the activations.
Example of Activation Tree
1 Introduction

2 Data Storage

3 Stack management
   - Stack allocation
     - Introduction
     - Activation tree
     - Control Stack and Activation record
     - Calling sequence
     - Variable-length data on the stack
     - Access to nonlocal data on the stack

4 Heap management
Procedure calls and returns are usually managed by a run-time stack called the control stack.

Each live activation has an activation record (or frame) on the control stack.

The entire sequence of activation records on the stack corresponding to the path in the activation tree to the activation where control currently resides.

The latter activation has its record at the top of the stack.
Example of Control Stack

- main
- quicksort(1,9)
- readArray
- partition(1,9)
- quicksort(1,3)
- quicksort(5,9)
- partition(1,3)
- partition(5,9)
- quicksort(1,0)
- partition(2,3)
- quicksort(2,3)
- quicksort(2,1)
- quicksort(3,3)
- partition(7,9)
- quicksort(5,9)
- quicksort(5,5)
- quicksort(7,7)
- quicksort(9,9)
The contents of activation records vary with the language being implemented; typically:

- Temporaries
- Local Data
- Saved Machine Status
- Access Link
- Control Link
- Returned Values
- Actual Parameters
The contents of activation records vary with the language being implemented; typically:

- Temporary values, such as those arising from the evaluation of expressions, in cases where the temporaries cannot be held in registers.
The contents of activation records vary with the language being implemented; typically:

- Local data belonging to the procedure whose activation record this is.
The contents of activation records vary with the language being implemented; typically:

- Information about the state of the machine just before the call to the procedure.
- It typically includes:
  - return address: the value of the ordinal counter to which the called procedure must return.
  - Registers: The contents of registers that were used by the calling procedure and that must be restored when the return occurs.
The contents of activation records vary with the language being implemented; typically:

- An “access link” may be needed to locate data needed by the called procedure but found elsewhere (in another activation record...)

```
<table>
<thead>
<tr>
<th>m+k</th>
</tr>
</thead>
<tbody>
<tr>
<td>Temporaries</td>
</tr>
<tr>
<td>Local Data</td>
</tr>
<tr>
<td>Saved Machine Status</td>
</tr>
<tr>
<td>Access Link</td>
</tr>
<tr>
<td>Control Link</td>
</tr>
<tr>
<td>Returned Values</td>
</tr>
<tr>
<td>Actual Parameters</td>
</tr>
</tbody>
</table>
```
The contents of activation records vary with the language being implemented; typically:

<table>
<thead>
<tr>
<th>m+k</th>
<th>Temporaries</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Local Data</td>
</tr>
<tr>
<td>Saved Machine Status</td>
<td>Access Link</td>
</tr>
<tr>
<td></td>
<td>Control Link</td>
</tr>
<tr>
<td>Returned Values</td>
<td>Actual Parameters</td>
</tr>
</tbody>
</table>

A “control link” is pointing to the activation record of the caller.
The contents of activation records vary with the language being implemented; typically:

- Space for the return value of the called function, if any.
- Not all called procedures return a value.
- We may prefer to place that value in a register for efficiency.
The contents of activation records vary with the language being implemented; typically:

- The actual parameters are given by the caller and used by the callee procedure.
- Commonly, these values are not placed in the activation record but rather in registers, when possible.
1 Introduction

2 Data Storage

3 Stack management
   - Stack allocation
     - Introduction
     - Activation tree
     - Control Stack and Activation record
     - Calling sequence
     - Variable-length data on the stack
     - Access to nonlocal data on the stack

4 Heap management

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
Definition (Calling Sequence)

A calling sequence is a code that allocates an activation record on the stack and enters information into its fields.

Definition (Return Sequence)

A return sequence is a code that deallocates an activation record from the stack and restores the state of the machine.

Calling sequences and the layout of activation records may differ greatly, even among implementations of the same language.

- The calling sequence is composed of:
  - Calling procedure (the “caller”),
  - Called procedure (the “callee”).
When designing calling sequences and the layout of the activation records, the following principles are used:

1. Values communicated between caller and callee are generally placed at the beginning of the callee activation record.

The caller can compute the actual parameters and put them at the top of the stack, without the necessity to create the entire record of the callee, and knowing how the callee’s record layout is. The caller knows where to put the return value, relative to its own record.
When designing calling sequences and the layout of the activation records, the following principles are used:

1. Fixed-length items are placed in the middle of the record.

If machine status are standardized, then programs such as debuggers will have an easier time deciphering the stack contents if an error occurs.
When designing calling sequences and the layout of the activation records, the following principles are used:

3. Items whose size may not be known early enough are placed at the end of the activation record.

Most of the variables have a size that can be determined by the compiler. But some cannot (dynamic arrays . . .)

The amount of space needed for temporaries is not known during the first phase of the intermediate code generation.
When designing calling sequences and the layout of the activation records, the following principles are used:

1. We must locate the “top-of-stack” pointer judiciously.

Commonly, it points to the end of the fixed-length fields in the activation record. The control link points to the “top-of-stack” of the previous record. Fixed-length data can then be accessed by fixed negative offset, and variable-length with a run-time positive offset.

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
1. The caller evaluates and stores the actual parameters.
2. The caller stores a return address.
   - It stores the old value of the top_sp into the callee’s activation record.
   - It increments top_sp to point to the callee activation record.
3. The callee saves the register values and other status information.
4. The callee initializes its local data and begins execution.
Typical Return Sequence

1. The callee places the return value next to the parameters.

2. Using information in the machine-status fields, the callee restores top_sp and other registers. It branches to the return address that the caller placed in the status field.

3. Although top_sp has been decremented, the caller knows where the return value is, relative to the current value of top_sp. The caller may use that value.
# Outline

1. **Introduction**
2. **Data Storage**
3. **Stack management**
   - Stack allocation
     - Introduction
     - Activation tree
     - Control Stack and Activation record
     - Calling sequence
     - Variable-length data on the stack
     - Access to nonlocal data on the stack
4. **Heap management**
• Programs contain a lot of data whose sizes are known at run-time; but which are local to a procedure.

• Because they are local to the procedure, they may be allocated on the stack.

• In most of the modern languages, these objects are allocated in the heap.

• However, it is also possible to allocate objects, arrays, or other data structures of unknown size on the stack.

**Why on the stack?**

Avoiding the expense of garbage collecting the space allocated for the variable-length data.
Below, the example of programs in C99 and C# in which a local array is declared. Its size depends on the value of the procedure parameter.

**The common strategy is to:**

1. Allocate the arrays at the end of the record.
2. Put pointers to the allocated regions in the local data.

```c
/* C99 */
void myFunction(int n) {
    float localArray[n];
    /* Do something with array */
}
```

```c
/* C# */
unsafe void myFunction(int n) {
    int* localArray = stackalloc int[size];
    /* Do something with array */
}
```
Two pointers are used:
- **top**: marks the actual top of the stack. It points to the position at which the next activation record will begin.
- **top_sp**: is used to find local, fixed-length fields of the top activation record.

When returning from a call:
$$\text{top} \leftarrow \text{top_sp} - \text{length(fixed_record_part)}$$
1 Introduction

2 Data Storage

3 Stack management
   - Stack allocation
     - Access to nonlocal data on the stack
       - Introduction
       - Data access without nested procedure
       - Issues with nested procedures
       - Nesting depth
       - Access links
       - Displays

4 Heap management

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
This section is devoted to the mechanisms of the procedures to access to their data.

Focusing on the mechanisms for finding data used within a procedure $p$ but that does not belong to $p$.

First, we study the cases of programs without nested functions.

Second, we introduces an algorithmic languages that permits to declare nested functions.
1 Introduction

2 Data Storage

3 Stack management
   - Stack allocation
   - Access to nonlocal data on the stack
     - Introduction
     - Data access without nested procedure
     - Issues with nested procedures
     - Nesting depth
     - Access links
     - Displays

4 Heap management
In languages similar to C, variables are declared:
- inside a single function, or
- outside any function (“globally”)

It is impossible to declare a procedure inside the scope of another procedures.

In such languages, allocation of storage for, and access to variables is simple:
- Global variables are allocated in the static storage. The locations remain fixed and are known at compile time.
- Any other name must be local to the activation at the top of the stack. The locations are relative to the top_sp pointer of the stack.
1 Introduction

2 Data Storage

3 Stack management
   - Stack allocation
   - Access to nonlocal data on the stack
     - Introduction
     - Data access without nested procedure
     - Issues with nested procedures
       - Nesting depth
       - Access links
       - Displays

4 Heap management
Many languages enable to declare procedures inside the scope of another procedure (Algol60, Pascal, ML, LISP).

LISP is a functional language: variables, after initialized cannot change.

Factorial Function, non-tail-recursion algorithm.

Factorial Function, tail-recursion algorithm.

(\texttt{deffun factorial} (n)
(\texttt{if} (\texttt{\leq} n 1)
  1
  (* n factorial (- n 1))))

(\texttt{deffun factorial} (n)
(\texttt{let} ((\texttt{deffun fact} (n, acc)
  (\texttt{if} (\texttt{\leq} n 1) acc
    (fact (- n 1) (* n acc))
  )))
  (fact n 1)))
Issue with Nested Procedures

With nested procedure declaration, it is far more complicated to determine the addresses of the names used in the procedure.

Example

- Let the procedure g declared inside the scope of the procedure p.
- g is accessing to the variable a, locally declared in p.
- It is difficult to determine at compile time where is the variable a in the stack, because of the recursive calls.
- The address of a in the stack can be determined only at run-time.

```plaintext
Procedure p(n)
begin
  Declare a ← n/2;
  Procedure g()
  begin
    if n > 1 then p(n-1);
    else if n = 1 then p(a/2);
  end
  g();
end
```
Example of the Issue

**Procedure** \( p(n) \)
begin
    Declare \( a = n \times 2; \)
    **Procedure** \( g() \)
begin
    if \( (n > 1) \) then \( p(n-1); \)
    else if \( (n = 1) \) then
        \( p(a/2); \)
    end
    \( g() \);
end
end
\( p(4) \)

**Activation Record Data**

- \( g \):
  - \( n \):
  - \( a \):
- \( p \):
  - \( n \):
  - \( a \):
- **main**
1 Introduction

2 Data Storage

3 Stack management
   - Stack allocation
   - Access to nonlocal data on the stack
     - Introduction
     - Data access without nested procedure
     - Issues with nested procedures
     - Nesting depth
     - Access links
     - Displays

4 Heap management

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
Informal Definition of Nesting Depth

- Let nesting depth 1 the declaration of a procedure outside another procedure.
- Let nesting depth 2 the declaration of a procedure inside one other procedure of nesting depth 1.
- Let nesting depth $n$ the declaration of a procedure inside one other procedure of nesting depth $n - 1$.

Procedure $p(n)$
begin
Declare a ← $n/2$;
Procedure $g()$
begin
if $n > 1$ then
$p(n-1)$;
else if $n = 1$ then
$p(a/2)$;
end
$g()$;
end

$p$ has nesting depth 1.

$g$ has nesting depth 2.
1 Introduction

2 Data Storage

3 Stack management
   ■ Stack allocation
     ■ Access to nonlocal data on the stack
       ■ Introduction
       ■ Data access without nested procedure
       ■ Issues with nested procedures
       ■ Nesting depth
     ■ Access links
     ■ Displays

4 Heap management
Access links provide a mean for the implementation of the static scope rule for nested functions.

- If procedure g is immediately nested in procedure p; then the access link in any activation of g points to the most recent activation of p.
- **Note:** nesting depth of p must be exactly one less than the nesting depth of g.

```plaintext
Procedure p(n)
begin
  Declare a ← n/2;
  Procedure g()
  begin
    if n > 1 then p(n-1);
    else if n = 1 then p(a/2);
  end
end
```
Access links form a chain from the activation record at the top of the stack to a sequence of activations at progressively lower nesting depths.

Along this chain are all the activations whose data and procedures are accessible to the currently executing procedure.
Example of Access Links

```
Procedure sqrt(q)
begin
    Procedure babylonian_algo(a,n)
    begin
        Declare a;
        b ← (a + q/a) / 2;
        if n > 0 then
            return b;
        else
            return babylonian_algo(a,n-1);
        end
    end
    return babylonian_algo(q/2, 10);
end
sqrt(q/2, 10);
```

Procedure sqrt(q)
begin
    Procedure babylonian_algo(a,n)
    begin
        Declare a;
        b ← (a + q/a) / 2;
        if n > 0 then
            return b;
        else
            return babylonian_algo(a,n-1);
        end
    end
    return babylonian_algo(q/2, 10);
end
sqrt(5);

Control Link
Access Link
4
Example of Access Links

Procedure $\sqrt{q}$
begin

Procedure babylonia_algo($a, n$)
begin

Declare $a$;
$b \leftarrow (a + q/a) / 2$;
if $n > 0$ then
  return $b$;
else
  return babylonia_algo($a, n-1$);
end

end

return babylonia_algo($q/2, 10$);
end

$q = 5$

Procedure $\sqrt{5}$;
**Example of Access Links**

```plaintext
Procedure sqrt(q)
begin
sqrt(q);
end

Procedure babylonian_algo(a,n)
begin
  Declare a;
  b ← (a + q/a) / 2;
  if n > 0 then
    | return b;
  else
    | return babylonian_algo(a,n-1);
  end
end

return babylonian_algo(q/2, 10);
```

**Procedure_sqrt(q)**

- **begin**
- **sqrt(q)**
- **end**
### Example of Access Links

To access to the value of `q`, we know at compile time, that it is reachable after one dereferencing in the access link pointer chain.

```plaintext
To access to the value of `q`, we know at compile time, that it is reachable after one dereferencing in the access link pointer chain.
```

---

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
Let the procedure $q$ calling $p$.

Let $N_\alpha$ the nesting depth of $\alpha$.

Let $D_\beta$ the set of the nesting procedures in which $\beta$ is defined.

**First Case**

$$(N_p > N_q) \Rightarrow (q \in D_p \land N_p = N_q + 1)$$

Then the access link from $p$ leads to $q$. 
Determining the Access Link Target

- Let the procedure q calling p.
- Let $N_\alpha$ the nesting depth of $\alpha$.
- Let $D_\beta$ the set of the nesting procedures in which $\beta$ is defined.

Second Case

$$(N_p \leq N_q) \Rightarrow \left( \exists r \mid \left( r \in D_p \wedge N_p = N_r + 1 \wedge r \in D_q \wedge N_r > N_q \right) \right)$$

Then
- The access link from p leads to r.
- There is $N_q - N_p + 1$ access links from q to r.
- Include recursive calls, where p = q.
A procedure p is passed to another procedure q as a parameter; q calls its parameter.

**Problem**
- If q does not know the context in which p appears in the program;
- it is impossible for q to know how to set the access link for p.

**Solution**
- The caller of a procedure with a procedure as parameter must also pass the proper access link to the parameter
- ie. the caller must pass the name and the access link as parameters.
Example of a Procedure Passing as Parameter

\[(\text{defun } a(x))\]
\[\text{(let ( (defun } b(f))\]
\[\text{ (\ldots f \ldots))}\]
\[\text{ (defun c(y))}\]
\[\text{(let ( (defun } d(z) (\ldots))))\]
\[\text{ (\ldots (b d) \ldots))}\]
\[\text{ )}\]
\[\text{ )}\]
\[\text{ )}\]
\[\text{ (\ldots (c 1) \ldots))}\]
\[\text{ )}\]
\[\text{ )}\]

Function \(a\) is called.
Example of a Procedure Passing as Parameter

Function \( c \) is called. According to the first case, access link leads to \( a \).
Example of a Procedure Passing as Parameter

Function `b` is called with the procedure `d` as parameter. According to the second case, access link leads to `a`. The context of `d` is also passed as parameter.

```lisp
(defun a(x)
    (let ((defun b(f)
            (... f ...))
          (defun c(y)
            (let ((defun d(z) (...)))
              (... (b d) ...))
          ))
    (... (c 1) ...)
)
```

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Access Link</th>
<th>Control Link</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a</code></td>
<td><code>x</code></td>
<td></td>
</tr>
<tr>
<td><code>b</code></td>
<td><code>f</code></td>
<td><code>&lt;d, P&gt;</code></td>
</tr>
<tr>
<td><code>c</code></td>
<td><code>y</code></td>
<td></td>
</tr>
<tr>
<td><code>d</code></td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
Example of a Procedure Passing as Parameter

Function d is called through the parameter f. The access link is directly taken from the context ①.
1 Introduction

2 Data Storage

3 Stack management
   - Stack allocation
   - Access to nonlocal data on the stack
     - Introduction
     - Data access without nested procedure
     - Issues with nested procedures
     - Nesting depth
     - Access links
     - Displays

4 Heap management

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
If the nesting depth gets large, we may have to follow long chains of links to reach the data we need.
Use of an auxiliary array \( d \), called the display.

- A display is a collection (an array) of pointers, one for each nesting depth.

- At all times, \( d[i] \) is a pointer to the highest activation record on the stack for any procedure at nesting depth \( i \).
If procedure p is executing, and it needs to access element x belonging to some procedure q, we need to look only in d[i], where i is the nesting depth of q.

The compiler knows what i is, so it can generate code to access x using d[i] and the offset of x from the top of the activation record for q.

The code never needs to follow a long chain of access links.
Each time a record is added in the stack, we need to save previous values of display entries.

- **If** procedure p at depth $N_p$ is called, **and**
- the activation record of p is not the first on the stack for a procedure at depth $N_p$, **then**

- Put the value of $d[N_p]$ in the activation record of p.
- Set $d[N_p]$ to the activation record of p.
Maintaining the Displays when Returning from Records

- Each time a record returns, we need to restore previous values of display entries.

- Set $d[N_p]$ to the value previously stored in the activation record of $p$. 
Example of Displays

(defun a(x)
  (let ( (defun b(f)
        (... f ...)
      )
    )
    (defun c(y)
      (let ( (defun d(z) (...)) )
        (... (b d) ...)
      )
    )
  )
)(... (c 1) ...)

The displays are pointing somewhere in the stack.
Function a is called.
Create the record.
Save d[1], which is pointing on a lower activation record.
Because $d[1]$ is not pointing to the record of a, change $d[1]$. 

(defun a(x)
  (let ((defun b(f)
        (… f …))
    (defun c(y)
      (let ((defun d(z) (…)) )
        (… (b d) …))
    )
  )
  (… (c 1) …)
)
Example of Displays

(defun a(x)
  (let ((defun b(f)
    (... f ...))
    (defun c(y)
      (let ((defun d(z) (...)) )
        (... (b d) ...)
      )
    )
  )
)

The function c is called.
Its record is created.
The previous value of d[2] is saved.
Because the record of `c` is not the one pointed by `d[2]`, set `d[2]` to leads to `c`.

```
(defun a(x)
  (let ((defun b(f)
        (… f …))
    (defun c(y)
      (let ((defun d(z) (…)) )
        (… (b d) …)
    )
  )
  (… (c 1) …)
)
```
Example of Displays

(defun a(x)
  (let ((
    (defun b(f)
      (... f ...) )
      )
    (defun c(y)
      (let ((
        (defun d(z) (... )) )
        (... (b d) ... )
      )
      )
    (... (c 1) ... )
  )
)
)

The function \texttt{b} is called. Save the \texttt{d[2]}, and set its value to leads to \texttt{b}. 
The function d is called through the parameter f.
The displays are updated.
Example of Displays

To obtain the value of x:

- Because x is at nesting depth 1, follows d[1] to reach the right record.
- Read the value of x in the record.
1 Introduction

2 Data Storage

3 Stack management

4 Heap management
   - Introduction
   - Memory manager
   - Locality in programs
   - Reduction of the fragmentation
   - Manual Deallocation

5 Garbage collection

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
The heap is the portion of the store that is used for data that lives indefinitely, or until the program explicitly deletes it.

Modern languages provides dedicated operators for the allocation and deallocation in the heap. For example, `new` and `delete` in C++.
This section describes the memory manager, the subsystem that allocates and deallocates space within the heap.

The memory manager is the interface between the application program and the operating system.

Garbage collection is the process of finding spaces within the heap that are no longer used by the program and can be reallocated. The garbage collector is an important subcomponent of the memory manager.
1. Introduction

2. Data Storage

3. Stack management

4. Heap management
   - Introduction
   - Memory manager
     - Locality in programs
     - Reduction of the fragmentation
     - Manual Deallocation

5. Garbage collection
The memory manager keeps track of all the free space in heap storage at all times.

Its two basic functions are:

1. allocation,
2. deallocation.
- Produces a chunk of contiguous memory for each variable or object associated to the allocation request.

- If not enough contiguous space is available for a chunk, it seeks to increase the heap storage space by requesting memory to the operating system.

- The defragmentation of the heap is generally not implemented.
Returns deallocated space to the pool of free space.

The deallocation space may be reused for future allocations.

Typically, the memory manager does not return memory to the operating system, even if the program’s heap usage drops.
Property: Space Efficiency

- A memory manager should minimize the total heap space need by a program.
- Space efficiency is achieved by minimizing the “fragmentation” (discussed later).
Property: Program Efficiency

- A memory manager should make good use of the memory subsystem to allow programs to run faster.
- The time taken to execute an instruction can vary widely depending on where objects are placed in memory.
- Programs tend to exhibit “locality” (discussed later), which refers to the nonrandom clustered way in which typical programs access memory.
- By attention to the placement of objects in memory, the memory manager can make better use of space, and make the program run faster.
Property: Low Overhead

- Because memory allocations and deallocations are frequent operations in many programs (such as ones written in Java), it is important that these operations be as efficient as possible.
- We wish to minimize the overhead, the fraction of execution time spent performing allocation and deallocation.

- Note: the overhead of allocation is dominated by a large amount of small requests; the overhead of managing large objects is less important.
The efficiency of a program is determined by:

1. the number of instructions executed, and
2. the time taken to execute each of these instructions.

Data-intensive programs can therefore benefit significantly from optimizations that make good use of the memory subsystem.

The run-time environment should prefer to use the memory storages close to the processor.

The concept of “locality” will help us to improve the use of the memory subsystem.
Introduction

Data Storage

Stack management

Heap management

Garbage collection

Conclusion
Hypothesis

The conventional wisdom is that programs spend 90% of their time executing 10% of the code.

In other words: they spend most of their time executing a relatively small fraction of the code and touching only a small fraction of the data.

Definition (Temporal Locality)

The memory locations, which are accessed by the program, are likely to be accessed again within a short period of time.

Definition (Spatial Locality)

The memory locations close to the accessed location are likely also to be accessed within a short period of time.
1. Programs often contain many instructions that are never executed.

2. After evolution, legacy systems contain many instructions that are no longer used.

3. Only a small fraction of the code that could be invoked is actually executed in a typical run of the program.

4. The typical program spends most of its time executing innermost loops and tight recursive cycles in a program.
Memory manager (and compiler optimizer) must be aware of how the operating system is managing its memory.

In modern systems, the memory is composed of several layers:

<table>
<thead>
<tr>
<th>Size</th>
<th>Type</th>
<th>Access Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; 2 GB</td>
<td>Virtual Memory</td>
<td>3-15 ms</td>
</tr>
<tr>
<td>256 MB-16GB</td>
<td>Physical Memory</td>
<td>100-150 ns</td>
</tr>
<tr>
<td>128 kB - 4MB</td>
<td>2\textsuperscript{nd}-Level Cache</td>
<td>40-60 ns</td>
</tr>
<tr>
<td>16-64kB</td>
<td>1\textsuperscript{st}-Level Cache</td>
<td>5-10 ns</td>
</tr>
<tr>
<td>32 Words</td>
<td>Registers</td>
<td>1 ns</td>
</tr>
</tbody>
</table>
Locality permits to take advantage of the memory hierarchy.

By placing the most common instructions and data in the fast-but-small storage,

While leaving the rest in the slow-but-large storage, we can lower the average memory-access time.
Optimization Policy

- Put the most-recent-used instruction in the fastest memory (example of spatial locality).
- Put together in the same memory page/block the instructions that may be always executed together (example of spatial locality).
- Temporal and spatial locality of data be improved by changing:
  - the data layout, or
  - the order of the computations.

Example

- For example, visiting a large amount of data and performing small operations on is not a good approach.
- Preferably, we should push down smaller data set into a faster memory level, and perform the computations on them.
At the beginning of the program, the heap is one contiguous unit of free space.

As the program allocates and deallocates memory, this space is broken up into free and used chunks.

The free chunks need not reside in a contiguous area of the heap.

The free chunks are named holes.

Alternating chunks and holes is named the fragmentation of the heap.
We reduce fragmentation by controlling how the memory manager places new objects in the heap.

Several approaches/strategies may be used:

1. **Best-Fit Object Placement**: Allocate the requested memory in the smallest available hole that is large enough. Not good for spatial locality.

2. **First-Fit Object Placement**: Allocate the requested memory in the first hole, which is able to contain the requested chunk. Less efficient than the previous one.

3. **Next-Fit Object Placement**: When no hole of the exact size was found, allocate the in the lastly split hole. Good for spatial locality and efficient.
To improve the implementation of the best-fit approach, we introduces the bins.

Free space chunks are grouped into bins, according to their sizes.

Many bins for the smaller sizes, because there are usually many more small objects.

Lea Memory Manager (GNU C compiler)

- Bins of every multiple of 8 bytes until 512 bytes.
- Larger-sized bins are logarithmically spaced.
- Within the bins the chunks are ordered by their sizes.
- Wilderness chunk: the largest bin because its size may be extended after requesting more memory to OS.
Let a requested chunk $c$ to be allocated.

1. If there is a bin for chunks of the size of $c$, we take any free space chunk from that bin.

2. If there is no bin for chunks of the size of $c$, we take the smallest bin that may include the requested size. Within that bin, we can use either a first-fit or a best-fit strategy. The remainder space of the selected chunk will generally be placed in a bin with smaller size.

3. If there is no more free chunk in a bin, we repeat the point 2 on a bin for a larger size; or we reach the wilderness chunk, which surely provide enough space.
When an object is deallocated, the memory manager makes its chunk free.

In some circumstances, it may also be possible to combine (coalesce) that freed chunk with adjacent chunks.

Two major data structures are basically used for supporting coalescing of adjacent free blocks:

1. Boundary Tags
2. Doubly Linked, Embedded Free List
When an object is deallocated, the memory manager makes its chunk free.

In some circumstances, it may also be possible to combine (coalesce) that freed chunk with adjacent chunks.

Two major data structures are basically used for supporting coalescing of adjacent free blocks:

1. **Boundary Tags**,  
2. **Doubly Linked Free List**, **Embedded Free List**.
- At both the low and high ends of a chunk, whether free or allocated, we keep vital information.

- **At both ends**, we put:
  1. a bit that indicates if the chunk is free or allocated.
  2. a count of the total number of bytes in the chunk.
The free chunks (but not the allocated ones) are linked in a doubly linked list.

In addition to the boundary tags, a pointer to the next and to the following free space chunks are added at the both ends of the chunk.

Does not need to allocate more space for these pointers: the pointers takes the unused bytes of the free chunks.

For the smaller chunks, they are expanded to allow to contain the pointers.
## Outline

1. Introduction
2. Data Storage
3. Stack management
4. Heap management
   - Introduction
   - Memory manager
   - Locality in programs
   - Reduction of the fragmentation
   - Manual Deallocation
5. Garbage collection

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
This subsection is devoted to the manual deallocation requests, such as in C and C++ languages.

Ideally, any storage that will no longer be accessed should be deleted.

Conversely, any storage that may be referenced must not be deleted.

It is hard to enforce these properties.
Main Problems with Manual Deallocation

Two common errors may occur in manual memory management:

1. **Memory Leak**: failing ever to delete data that cannot be referenced.

2. **Dangling-pointer reference**: referencing deleted data.
Problem of Memory Leak

Observation

- It is hard for a developer to tell if a program will never refer to some storage in the future.
- The common mistake is not deleting storage that will never be referenced.

Problem

May slow down the execution of the program due to increased memory usage.

Remarks

- Correctness of the program is not changed.
- Many programs may tolerate leaks but not the long-time and the critical ones (operating systems, server code...).
Automatic garbage collection gets rid of memory leaks by deallocating all the garbage. Even with a garbage collector, programs may still use more memory than necessary.

A programmer may know that an object will never be referenced. He must deliberately remove the references to objects that will never be referenced, so the objects can be deallocated automatically.
Problem of Dangling-Pointer Reference

Observation

- Deletion of a storage, and then referencing the deleted storage.
- These pointers are named “dangling pointers.”

Problem

- When the storage has been reallocated, it produces random effects on the program.
- Writing through a dangling pointer changes an other variable than the one expecting by the dangling pointer.

Remarks

Read, write or deallocate a pointer is named “dereferencing the pointer.”
Unfortunately, there is no turnkey solution.

1. The programmer must be aware and may pay attention to his uses of the pointers.

2. The dangling-pointer-dereference error does not occur in run-time environments that have an automatic garbage collector.
Illegal Address Error

Definition

This error occurs when the address to dereference is null or outside the bounds of any allocated memory (including the bounds of the memory space of the process).

- The illegal address error is related to the dangling-pointer-dereference error.
- This error is at the origin of many security violations from hackers.
- One solution is that the compiler inserts checks with every access, to make sure it is within the bounds.
- The compiler optimizer may remove several of these checks when they are detected as not necessary.
Object Ownership

- Associate an owner with each object at all times.
- The owner is usually a function.
- The owner is responsible for either deleting the object or for passing the object to another owner.
- Non-owning points may reference the object, but the object must never be deallocated through them.
- This convention eliminates memory leaks, and the deletion of the same object twice.
- This convention does not solve the dangling-point-reference problem.
Reference Counting

- Associate a count with each dynamically allocated object.
- Whenever the reference to the object is created, the counter is incremented.
- Whenever the reference to the object is deleted, the counter is decremented.
- The storage for the object is released when the counter is zero.
- Expensive operation.
- Do not work with inaccessible circular data structures.
Region-Based Allocation

- When objects are created to be used only within some step of a computation, we can allocate all such objects in the same region.
- We then delete the entire region once that computation step completes.
- Limited applicability.
- Very efficient.
2 Data Storage

3 Stack management

4 Heap management

5 Garbage collection
   - Introduction
   - Properties of a garbage collector
   - Reachability of data
   - Reference-counting garbage collector
   - Trace-based garbage collector
   - Short-pause garbage collector

6 Conclusion
Data that cannot be referenced as named garbage.

Many high-level programming languages remove the burden of manual memory management from the programmer by offering automatic garbage collection.

The garbage collection (GC) is the process to deallocate no-more referenced storages from the heap.

The first garbage collection dates from the initial implementation of LISP in 1958.

Other languages provide natively a GC: Java, Perl, ML, Modula-3, Prolog, Smalltalk, C#, Ruby, Python...
Hypothesis for Garbage Collection

The garbage collector must know the type of the objects at run-time. This type permits to determine:

1. the size of the object in bytes.
2. its components that are references to other objects.

The references to the objects are always to the address of the beginning of these objects.

All the references to the same object have the same value and may be identified easily.
Definition (Mutator)

The mutator, the user program, modifies the collection of objects in the heap.

- The mutator creates objects by acquiring space from the memory manager.
- The mutator may introduce and drop references to existing objects.
- Objects become garbage when the mutator program cannot “reach” them.
- The GC finds the unreachable objects and reclaims their space by handling them to the memory manager.
Outline

2 Data Storage

3 Stack management

4 Heap management

5 Garbage collection
   - Introduction
   - Properties of a garbage collector
     - Reachability of data
     - Reference-counting garbage collector
     - Trace-based garbage collector
     - Short-pause garbage collector

6 Conclusion
For a GC to work, the language must be type safe.

The type of the data may be determined at compile or run-time.

GC must be able to tell whether any given data element or component of a data element is, could be used as, a pointer to a chunk.

Two family of type-safe languages:

1. **Statically typed languages**: the types are determined at compile time (ML...)
2. **Dynamically typed languages**: the types are determined at run-time (Java...)
Unsafe languages (C, C++...) are bad candidate for GC.

In unsafe languages, memory addresses can be manipulated arbitrarily (pointer arithmetic...)

Thus programs can refer to any location in memory at any time.

Consequently, no memory location can be considered to be inaccessible, and no storage can ever be reclaimed safely.
**Performance Metrics**

- **Overall Execution Time:** It is important that GC is not significantly increase the total run time of an application.

- **Space Usage:** The GC must avoid fragmentation and make the best use of the available memory.

- **Pause Time:** Simple GC causes the mutator to pause suddenly for an extremely long time. The maximum pause time must be minimized.

- **Program Locality:**
  - The speed of the GC cannot be evaluated solely by its running time.
  - The GC controls the placement of data and thus influences the data locality of the mutator program.
  - GC can improve the mutator’s temporal locality by freeing the space and reusing it.
  - It can improve the mutator’s spatial locality by relocating data used together in the same cache or pages.
Introduction
Data Storage
Stack management
Heap management
Garbage collection
Conclusion

2 Data Storage

3 Stack management

4 Heap management

5 Garbage collection
  ■ Introduction
  ■ Properties of a garbage collector
  ■ Reachability of data
  ■ Reference-counting garbage collector
  ■ Trace-based garbage collector
  ■ Short-pause garbage collector

6 Conclusion

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
**Definition**

It refer to all the data that can be accessed directly by a program, without having to dereference any pointer.

**Example**

In Java, the root set is composed of all the static fields and all the variables in the stack.

- A program can reach any member of its root set at any time.
- Recursively, any object with a reference that is stored in the field members or array elements of any reachable object is itself reachable.
- **Note:** when an object becomes unreachable, it will never be reachable again.
Basic Operations to Change the Root Set (#1)

Object allocation

- Performed by the memory manager.
- The Memory manager returns a reference to each newly allocated chunk of memory.
- This operation adds members to the set of reachable objects.

Parameter passing and return values

- References to objects are passed from the actual input parameter to the corresponding formal parameters; and from the returned result back to the caller.
- Objects pointed to by these references remain reachable.
Reference assignments

- Assignments of the $x = y$ ($x$ and $y$ are references) have two effects:
  1. $x$ is now a reference to the object referred by $y$. The object referenced by $x$ and $y$ is reachable while $x$ or $y$ is reachable.
  2. The original reference of $x$ is lost. If this lost reference is the last on the object, the object becomes unreachable.

- When an object becomes unreachable, all the reachable objects inside becomes also unreachable.
Procedure returns

- As a procedure exists, the frame holding its local variables is popped off the stack.

- If the frame holds the only reachable reference to any object, that object becomes unreachable.

- If the now unreachable objects holds the only references to other objects, they too become unreachable, and so on.
Transitions from reachability to unreachability are caught, or the reachable objects are periodically located; assuming that all the other objects are not reachable.

Reference counting is an approximation of the first approach:

- A count of the reference to an object is maintained.
- When the count goes to zero, the object becomes unreachable (discussed in the next section).
The reachability is computed by tracing all the references transitively.

- A trace-based garbage collector starts by labeling, marking, all objects in the root set as “reachable.”
- It examines iteratively all the references in reachable objects to find more reachable objects.
- It labels the discovered objects as “reachable.”
- Once the reachable set is computed, it may find the unreachable objects.
- All the unreachable objects could be deallocated at the same time.
Outline

2 Data Storage

3 Stack management

4 Heap management

5 Garbage collection
   - Introduction
   - Properties of a garbage collector
   - Reachability of data
   - Reference-counting garbage collector
   - Trace-based garbage collector
   - Short-pause garbage collector

6 Conclusion

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
This section is devoted to a simple and imperfect garbage collector based on reference counting.

With a reference-counting garbage collector, every object must have a field for the reference count; and maintains as described in the following slides.
Object allocation: The reference count of the new object is set to 1.

Parameter Passing: The reference count of each object passed into a procedure is incremented.

Reference Assignments: For statement $u = v$ ($u$ and $v$ are references), the reference count of the object referred to by $v$ goes up by one, and the count for the old object referred to by $u$ goes down by one.
**Procedure Returns:** As a procedure exists, objects referred to by the local variables in its activation record have their counts decremented. If several local variables hold references to the same object, that object’s count must be incremented once for each such reference.

**Transitive Loss of Reachability:** Whenever the reference count of an object becomes zero, we must also decrement the count of each object pointed to by a reference within the object.
Example of Reference-Counting Garbage Collector

```
class Obj {
    public Obj a = null;
    public Obj b = null;
}
class Main {
    public static void main(String[] args) {
        Obj o1 = new Obj();
        {
            Obj o2 = new Obj();
            Obj o3 = new Obj();
            o1.b = o2;
            o2.a = o1;
            o2.b = o3;
            o3.b = o1;
        }
    }
}
```

// root = stack+static
Example of Reference-Counting Garbage Collector

```java
class Obj {
    public Obj a = null;
    public Obj b = null;
}

class Main {
    public static void main(String[] args) {
        Obj o1 = new Obj();
        {
            Obj o2 = new Obj();
            Obj o3 = new Obj();
            o1.b = o2;
            o2.a = o1;
            o2.b = o3;
            o3.b = o1;
        }
    }
}
```

refs=1

root = stack+static

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
Example of Reference-Counting Garbage Collector

```java
class Obj {
    public Obj a = null;
    public Obj b = null;
}
class Main {
    public static void main(String[] args) {
        Obj o1 = new Obj();
        {
            Obj o2 = new Obj();
            Obj o3 = new Obj();
            o1.b = o2;
            o2.a = o1;
            o2.b = o3;
            o3.b = o1;
        }
    }
}
```

```
root = stack+static
Refs=1
```

```
Ref=1
```

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr

Université de Technologie Belfort-Montbéliard
Example of Reference-Counting Garbage Collector

class Obj {
  public Obj a = null;
  public Obj b = null;
}
class Main {
  public static void main(String[] args) {
    Obj o1 = new Obj();
    {
      Obj o2 = new Obj();
      Obj o3 = new Obj();
      o1.b = o2;
      o2.a = o1;
      o2.b = o3;
      o3.b = o1;
    }
  }
}
Example of Reference-Counting Garbage Collector

```java
class Obj {
    public Obj a = null;
    public Obj b = null;
}
class Main {
    public static void main(String[] args) {
        Obj o1 = new Obj();
        {
            Obj o2 = new Obj();
            Obj o3 = new Obj();
            o1.b = o2;
            o2.a = o1;
            o2.b = o3;
            o3.b = o1;
        }
    }
}
```

(root = stack+static)
Example of Reference-Counting Garbage Collector

class Obj {
    public Obj a = null;
    public Obj b = null;
}
class Main {
    public static void main(String[] args) {
        Obj o1 = new Obj();
        {
            Obj o2 = new Obj();
            Obj o3 = new Obj();
            o1.b = o2;
            o2.a = o1;
            o2.b = o3;
            o3.b = o1;
        }
    }
}
Example of Reference-Counting Garbage Collector

```java
class Obj {
    public Obj a = null;
    public Obj b = null;
}
class Main {
    public static void main(String[] args) {
        Obj o1 = new Obj();
        {
            Obj o2 = new Obj();
            Obj o3 = new Obj();
            o1.b = o2;
            o2.a = o1;
            o2.b = o3;
            o3.b = o1;
        }
    }
}
```

Diagram:
- `root = stack+static`
- `Refs=2`
- `o1`, `o2`, `o3`
Example of Reference-Counting Garbage Collector

class Obj {
    public Obj a = null;
    public Obj b = null;
}
class Main {
    public static void main(String[] args) {
        Obj o1 = new Obj();
        {
            Obj o2 = new Obj();
            Obj o3 = new Obj();
            o1.b = o2;
            o2.a = o1;
            o2.b = o3;
            o3.b = o1;
        }
    }
}
Example of Reference-Counting Garbage Collector

```java
class Obj {
    public Obj a = null;
    public Obj b = null;
}
class Main {
    public static void main(String[] args) {
        Obj o1 = new Obj();
        {
            Obj o2 = new Obj();
            Obj o3 = new Obj();
            o1.b = o2;
            o2.a = o1;
            o2.b = o3;
            o3.b = o1;
        }
    }
}
```
Example of Reference-Counting Garbage Collector

```java
class Obj {
    public Obj a = null;
    public Obj b = null;
}
class Main {
    public static void main(String[] args) {
        Obj o1 = new Obj();
        {
            Obj o2 = new Obj();
            Obj o3 = new Obj();
            o1.b = o2;
            o2.a = o1;
            o2.b = o3;
            o3.b = o1;
        }
    }
}
```

Refs=2

Refs=1

Refs=1

root = stack+static
This set of objects should be garbage collected. But their counters are greater than 0. Such a situation is tantamount to a memory leak, since this set of objects will never be deallocated.
A line is added to reset the reference from $o1$ to $o2$. 

```java
class Obj {
    public Obj a = null;
    public Obj b = null;
}
class Main {
    public static void main(String[] args) {
        Obj o1 = new Obj();
        {
            Obj o2 = new Obj();
            Obj o3 = new Obj();
            o1.b = o2;
            o2.a = o1;
            o2.b = o3;
            o3.b = o1;
        }
        o1.b = null;
    }
}
```
Solving Memory Leak with Reference-Counting Garbage Collector

```java
class Obj {
    public Obj a = null;
    public Obj b = null;
}
class Main {
    public static void main(String[] args) {
        Obj o1 = new Obj();
        {
            Obj o2 = new Obj();
            Obj o3 = new Obj();
            o1.b = o2;
            o2.a = o1;
            o2.b = o3;
            o3.b = o1;
        }
        o1.b = null;
    }
}
```

root = stack+static

Refs=3

Refs=1

Refs=0

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
Solving Memory Leak with Reference-Counting Garbage Collector

```java
class Obj {
    public Obj a = null;
    public Obj b = null;
}
class Main {
    public static void main(String[] args) {
        Obj o1 = new Obj();
        {
            Obj o2 = new Obj();
            Obj o3 = new Obj();
            o1.b = o2;
            o2.a = o1;
            o2.b = o3;
            o3.b = o1;
        }
        o1.b = null;
    }
}
```

root = stack+static

Refs=2

Refs=1

Refs=0
The chunk, previously referred by o2, is no more referenced. It is garbage collected.

```
class Obj {
    public Obj a = null;
    public Obj b = null;
}
class Main {
    public static void main(String[] args) {
        Obj o1 = new Obj();
        {
            Obj o2 = new Obj();
            Obj o3 = new Obj();
            o1.b = o2;
            o2.a = o1;
            o2.b = o3;
            o3.b = o1;
        }
        o1.b = null;
    }
}
```
Solving Memory Leak with Reference-Counting Garbage Collector

```java
class Obj {
    public Obj a = null;
    public Obj b = null;
}
class Main {
    public static void main(String[] args) {
        Obj o1 = new Obj();
        {
            Obj o2 = new Obj();
            Obj o3 = new Obj();
            o1.b = o2;
            o2.a = o1;
            o2.b = o3;
            o3.b = o1;
        }
        o1.b = null;
    }
}
```

Chunk refered by o3 is garbage collected.
Solving Memory Leak with Reference-Counting Garbage Collector

Chunk refered by o1 is garbage collected. There is no memory leak.

```java
class Obj {
    public Obj a = null;
    public Obj b = null;
}
class Main {
    public static void main(String[] args) {
        Obj o1 = new Obj();
        {
            Obj o2 = new Obj();
            Obj o3 = new Obj();
            o1.b = o2;
            o2.a = o1;
            o2.b = o3;
            o3.b = o1;
        }
        o1.b = null;
    }
}
```

---

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
The concept of deferred reference counting has been proposed as a mean to eliminate the overhead associated with updating the reference counts due to stack accesses.

Reference counts do not include references from the root set of the program.

An object is not considered to be garbage until the entire root set is scanned and no reference to the object is found.
Advantages of Reference Counting

1. Garbage Collection is performed in an incremental fashion.
   - The operations are made through the mutator’s operations.
   - Removing one reference may render a large number of objects unreachable, the operation of recursively modifying reference counts can easily be deferred and performed piecemeal across time.
   - Reference counting is particularly attractive when timing deadlines must be met.

2. Garbage are collected immediately, keeping space usage low.
Disadvantages of Reference Counting

1. Reference counting cannot collect unreachable, cyclic data structures.
   - Cyclic data structures are quite plausible.
   - Data structures often point back to their parent nodes, or point to each other as cross references.

2. Overhead of reference counting is high:
   - Additional operations were introduced with each reference assignment.
   - Additional operations were introduced with each procedure call and exit.
   - The overhead is proportional to the amount of computation in the program, and not just to the number of objects in the system.
4 Heap management

5 Garbage collection
  ■ Introduction
  ■ Properties of a garbage collector
  ■ Reachability of data
  ■ Reference-counting garbage collector
  ■ Trace-based garbage collector
    ■ Introduction
    ■ Basic mark-and-sweep collector
    ■ Mark-and-compact garbage collector
    ■ Copying collector
    ■ Brief Comparison
  ■ Short-pause garbage collector

6 Conclusion
Instead of collecting garbage as it is created, trace-based collectors run periodically to find unreachable objects.

Typically, we run the trace-based collector whenever the free space is exhausted or its amount drops below some threshold.

All trace-based algorithms:

1. compute the set of reachable objects, and
2. Take the complement of this list.
States of the Chunks

**Definition (Free)**
- A chunk is in the **Free** state if it is ready to be allocated.
- A Free chunk must not hold a reachable object.

**Definition (Allocated)**
- A chunk is in the **Allocated** state if it was used to store data.
- An allocated chunk must be in one of the three substates:
  1. Unreached
  2. Unscanned
  3. Scanned

allocate

Free

deallocate

Allocated
Definition (Unreached)

- Chunks are presumed unreachable, unless proven reachable by tracing.
- A chunk is in the **Unreached** state at any point during garbage collection if its reachability has not yet been established.
- After a round of garbage collection, the state of a reachable object is reset to Unreachable to get ready for the next round (see the next states).
**Definition (Unscanned)**

- A chunk is in the **Unscanned** state if it is known as reachable, but its pointers have not yet been scanned.

- The transition to Unscanned from Unreached occurs when we discover that a chunk is reachable.
States of an Allocated Chunk

Definition (Scanned)

- Every Unscanned object will eventually be scanned and transition to the **Scanned** state.
- To scan an object, we examine each of the pointers within it and follow those pointers to the objects to which they refer.
- A scanned object can only contain references to other scanned or unscanned objects, never to unreached objects. Consequently, the accessible chunks are moved to the Unscanned state if they are unreachable.
At the end of its algorithm, the GC is deallocating the unreached chunks.

It is setting the states of the reached chunks to “Unreached” for the next GC execution.
Outline

4 Heap management

5 Garbage collection
- Introduction
- Properties of a garbage collector
- Reachability of data
- Reference-counting garbage collector
- Trace-based garbage collector
  - Introduction
  - Basic mark-and-sweep collector
  - Mark-and-compact garbage collector
  - Copying collector
  - Brief Comparison
- Short-pause garbage collector

6 Conclusion
Mark-and-sweep collectors are "stop-the-world" algorithms. They find all the unreachable objects, and put them on the list of free space.

The algorithm visits and "marks" all the reachable objects in the first tracing step. Then it "sweeps" the entire heap to free up unreachable objects.
**Algorithm of the Mark-and-Sweep Collector (#1)**

**Input**: A root set of objects, a heap, and a free list (named Free), with all the unallocated chunks of the heap.

**Output**: A modified Free list after all the garbage has been removed.

```plaintext
begin
    /* MARKING PHASE */
    Unscanned ← copy-of(root);
    foreach o ∈ Unscanned do
        reached_bit[o] ← false;
    end
    while ∃ o ∈ Unscanned do
        Unscanned ← Unscanned \{o};
        foreach r ∈ references_in(o) do
            if neg reached_bit[r] then
                reached_bit[r] ← true;
                Unscanned ← Unscanned ∪ {r};
            end
        end
    end
end
```

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
Algorithm of the Mark-and-Sweep Collector (#2)

```c
/* SWEEPING PHASE */
Free ← ∅;
foreach c ∈ chunks do
    if neg reached_bit[c] then
        Free ← Free ∪ {c};
    else
        reached_bit[c] ← false;
    end
end
```

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
Problem

- The final step in the mark-and-sweep algorithm is expensive.
- Not easy way to find the unreachable objects without examining the entire heap.

Improved Algorithm

- **Baker’s Mark-and-sweep Algorithm.**
- It keeps a list of all allocated objects and compute a difference between the allocated objects and the reached objects.
[Baker's Algorithm of the Mark-and-Sweep Collector (#1)]

Input: A root set of objects, a heap, a free list (named Free), a list of allocated objects Unreached.

Output: A modified Free and Unreached lists.

begin

/* MARKING PHASE */
Unscanned ← ∅;
Scanned ← ∅;

foreach o ∈ root ∩ Unreached do
    Unreached ← Unreached \{o\}; Unscanned ← Unscanned ∪{o};
end

while ∃ o ∈ Unscanned do
    Unscanned ← Unscanned \{o\};
    Scanned ← Scanned ∪{o};
    foreach r ∈ references in(o) do
        if r ∈ Unreached then
            Unreached ← Unreached \{r\};
            Unscanned ← Unscanned ∪{r};
        end
    end
end
Baker’s Algorithm of the Mark-and-Sweep Collector (#2)

/* SWEEPING PHASE */
Free ← Free ∪ Unreached;
Unreached ← Scanned;
4 Heap management

5 Garbage collection
   - Introduction
   - Properties of a garbage collector
   - Reachability of data
   - Reference-counting garbage collector
   - Trace-based garbage collector
     - Introduction
     - Basic mark-and-sweep collector
     - Mark-and-compact garbage collector
     - Copying collector
     - Brief Comparison
   - Short-pause garbage collector

6 Conclusion

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
Relocating collectors move reachable objects around in the heap to eliminate memory fragmentation.

After identifying all the holes, instead of freeing them, one alternative is to relocate the allocated objects in one end of the heap.

The rest of the memory becomes a single free chunk.

Two major approaches for building a relocating collector are:

1. A mark-and-compact collector
2. A copying collector
The mark-and-compact collector follows:

1. **Marking Phase**: similar to the mark-and-sweep algorithms

2. **Object Relocation**:
   - The allocated regions of the heap are scanned.
   - The address of each reachable object is computed from the low end of the heap.
   - The addresses are stored in a structure named NewLocation.

3. **Object Copy**:
   - The objects are copied to their new locations,
   - The references in the objects to point to are updated.
Algorithm of the Mark-and-Compact Collector (#1)

Input: A root set of objects, a heap, a pointer marking the start of the free space (named Free).
Output: The new value of pointer Free

begin

/* MARKING PHASE */
Unscanned ← copy_of(root);
foreach o ∈ Unscanned do
    reached_bit[o] ← false;
end

while ∃ o ∈ Unscanned do
    Unscanned ← Unscanned \ {o};
    foreach r ∈ references_in(o) do
        if neg reached_bit[r] then
            reached_bit[r] ← true;
            Unscanned ← Unscanned ∪ {r};
        end
    end
end

end
Algorithm of the Mark-and-Compact Collector (#2)

```plaintext
/* COMPUTE THE NEW LOCATIONS */
NewLocation ← [];
Free ← first address in the heap; foreach c ∈ chunks[0..] do
    if reached_bit[c] then
        NewLocation[c] ← Free;
        Free ← Free size_of(c);
    end
end
```

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
/* RETARGET THE REFERENCES AND MOVE REACHED OBJECTS */

foreach \( c \in \text{chunks}[0..] \) do
  if \( \text{reached\_bit}[c] \) then
    foreach \( r \in \text{references\_in}(c) \) do
      \( c.r \leftarrow \text{NewLocation}[c.r] \);
    end
    Copy \( c \) to \( \text{NewLocation}[c] \);
  end
end

foreach \( r \in \text{references\_in}(\text{root}) \) do
  \( r \leftarrow \text{NewLocation}[r] \)
end
4 Heap management

5 Garbage collection
- Introduction
- Properties of a garbage collector
- Reachability of data
- Reference-counting garbage collector
- Trace-based garbage collector
  - Introduction
  - Basic mark-and-sweep collector
  - Mark-and-compact garbage collector
- Copying collector
- Brief Comparison
- Short-pause garbage collector
A copying collector reserves space to which the objects can move.

The memory space is partitioned into two semispaces $A$ and $B$.

The mutator allocates in $A$ until it fill up.

Then the mutator is stopped and the GC copies the reachable objects to $B$.

When GC finished, the roles of $A$ and $B$ are reversed.

The algorithm is due to C.J. Cheney.
4 Heap management

5 Garbage collection

- Introduction
- Properties of a garbage collector
- Reachability of data
- Reference-counting garbage collector
- Trace-based garbage collector
  - Introduction
  - Basic mark-and-sweep collector
  - Mark-and-compact garbage collector
  - Copying collector
  - Brief Comparison
- Short-pause garbage collector
Comparing the Costs

- **Basic Mark-and-sweep**: Proportional to the number of chunks in heap.

- **Baker’s Mark-and-sweep**: Proportional to the number of reached objects.

- **Basic Mark-and-compact**: Proportional to the number of chunks in the heap plus the total size of the reached objects.

- **Cheney’s Copying Collector**: Proportional to the total of the reached objects.
4 Heap management

5 Garbage collection
   - Introduction
   - Properties of a garbage collector
   - Reachability of data
   - Reference-counting garbage collector
   - Trace-based garbage collector
   - Short-pause garbage collector
     - Introduction
     - Incremental garbage collector
     - Partial garbage collector

6 Conclusion
Problem of the trace-based collectors

- Trace-based collectors do stop-the-world GC.
- It may introduce long pauses into execution of user programs.

First Solution: Incremental Collection
Divide the work in time, by interleaving GC with the mutation.

Second Solution: Partial Collection
Divide the work in space, by collecting a subset of the garbage at a time.
4 Heap management

5 Garbage collection

- Introduction
- Properties of a garbage collector
- Reachability of data
- Reference-counting garbage collector
- Trace-based garbage collector
- Short-pause garbage collector
  - Introduction
  - Incremental garbage collector
  - Partial garbage collector

6 Conclusion
Incremental collectors are conservative:
- While GC must not collect objects that are not garbage,
- It does not have to collect all the garbage in each round.

The garbage left in memory is named floating garbage.
Incremental collectors overestimate the set of reachable objects.
Incremental Algorithm

1. The program’s root set is processed automatically, without interference with the mutator.

2. After finding the initial set of unscanned objects, the mutator’s actions are interleaved with the tracing step.

3. During this period, any of the mutator’s actions that may change reachability are recorded succinctly, in a side table.

4. The side table is used by the collector to adjust the memory allocation when the mutator’s actions resume their execution.

5. If there is not enough memory space, the collector blocks the mutator until it finished to collect the garbage.
The set of reachable objects when tracing finished is:

\[(R \cup \text{New}) \setminus \text{Lost}\]

- \(R\): the set of reachable objects at the beginning of garbage collection.
- \(\text{New}\): the set of allocated objects during garbage collection.
- \(\text{Lost}\): the set of objects that have become unreachable due to lost references.
It is expensive to compute an object’s reachability every time.

Incremental collectors do not attempt to collect all the garbages at the end of the tracing.

Every garbage left behind (floating garbage) should be a subset of the Lost objects.

\[ (R \cup \text{New}) \setminus \text{Lost} \subseteq S \subseteq (R \cup \text{New}) \]
First, we use a tracing algorithm to find the upper bounds of $R \cup \text{New}$. 

The behavior of the mutator is modified during the tracing:
- All references that existed before GC are preserved.
- All objects created are considered reachable immediately and are placed in the Unscanned state.

This scheme is conservative and finds $R$ and $\text{New}$.

But the cost is high because the algorithm intercept all the write operations and remembers all the overwritten references.

The following slides proposes a solution.
If mutator and tracing GC algorithm are interleaved, then some reachable objects may be misclassified as unreachable.

Because the mutator may violate the following invariant of the GC algorithm:
A scanned object can only contain references to other scanned or unscanned objects, never unreached objects.
Examples of Violation

- The garbage collector finds object $A$ reachable and scans the pointers within $A$, thereby putting $A$ in the Scanned state.

- The mutator stores a reference to an Unreached (but reachable) object $B$ into the Scanned object $A$. It does so by copying a reference to $B$ from an object $C$ that is currently in the Unreached or Unscanned state.

- The mutator loses the reference to $B$ in object $C$. It may have overwritten $C$’s reference to $B$ before the reference is scanned, or $C$ may have become unreachable and never have reached the Unscanned state to have its reference scanned.
The garbage collector finds object \( A \) reachable and scans the pointers within \( A \), thereby putting \( A \) in the Scanned state.

The mutator stores a reference to an Unreached (but reachable) object \( B \) into the Scanned object \( A \). It does so by copying a reference to \( B \) from an object \( C \) that is currently in the Unreached or Unscanned state.

The mutator loses the reference to \( B \) in object \( C \). It may have overwritten \( C \)'s reference to \( B \) before the reference is scanned, or \( C \) may have become unreachable and never have reached the Unscanned state to have its reference scanned.
Examples of Violation

- The garbage collector finds object A reachable and scans the pointers within A, thereby putting A in the Scanned state.

- The mutator stores a reference to an Unreached (but reachable) object B into the Scanned object A. It does so by copying a reference to B from an object C that is currently in the Unreached or Unscanned state.

- The mutator loses the reference to B in object C. It may have overwritten C’s reference to B before the reference is scanned, or C may have become unreachable and never have reached the Unscanned state to have its reference scanned.
Avoiding the Violations (#1)

- **Write Barriers:**
  - Intercepts writes of references into a Scanned object $A$.
  - When the reference is to an Unreached object $B$, then classify the object $B$ as reachable and place it in an Unscanned state;
  - or put the object $A$ in an Unscanned state.

- **Read Barriers:**
  - Intercept the reads of references in Unreached or Unscanned objects.
  - When the mutator reads the reference to object $A$ from an object in Unreached or Unscanned state, classify $A$ as reachable and put it in the Unscanned state.
Transfer Barriers:

- Intercept the loss of the original reference in an Unreached or Unscanned object.
- When the mutator overwrites a reference in an Unreached or Unscanned object, save the reference being overwritten, classify it as reachable, and place the reference itself in the Unscanned state.
Comparison of the Mutator Barriers

- Write barriers are the most efficient of the barriers.

- Read barriers are more expensive because typically there are many more reads than there are writes.

- Transfer barriers are not competitive; because many objects “die young,” this approach would retain many unreachable objects.
4 Heap management

5 Garbage collection
   - Introduction
   - Properties of a garbage collector
   - Reachability of data
   - Reference-counting garbage collector
   - Trace-based garbage collector
   - Short-pause garbage collector
     - Introduction
     - Incremental garbage collector
     - Partial garbage collector

6 Conclusion
Fact: “Objects typically die young.”

These objects become unreachable before the GC is invoked.

Consequently, GC is cost effective with the approaches presented in the previous slides.

The same “mature” objects were found and copied at every round of the GC.

Two major approaches make partial GC to be more effective:

1. Generational GC
2. Train Algorithm
General Principle of the Generational GC

- The heap is divided in partitions: 0, 1, ..., n (0 is for the younger data)

- Objects are created in partition 0.

- When the partition 0 fills up, it is GC and the reachable objects are moved in partition 1.

- The same algorithm is used between partition 1 and 2, 2 and 3...
■ The train algorithm uses fixed-length partitions, called cars.

Definition (Car)

■ A car might be a single disk block, assuming there are no object larger than disk blocks; or
■ the car size could be larger, but it is fixed once and for all.

Definition (Train)

Cars are organized into trains. There is no limit to the number of cars in a train, and no limit to the number of trains.
Two approaches to collect the garbages:

1. The first car in lexicographic order is collected in one incremental garbage-collection step.
   - This step is similar to collection of the first partition in the generational algorithm, since we maintain a “remembered” list of all points from outside the car.
   - We identify objects with no references at all, as well as garbage cycles that are contained completely within this car.
   - Reachable objects in the car are always moved to some other car, so each garbage-collected car becomes empty and can be removed from the train.

2. Sometimes, the first train has no external reference.
   - That is, there are no pointers from the root set to any car of the train, and the remembered sets for the cars contain only references from other cars in the train, not from other trains.
   - In this situation, the train is a huge collection of cyclic garbage, and we delete the entire train.
Two approaches to collect the garbages:

1. The first car in lexicographic order is collected in one incremental garbage-collection step.
   - This step is similar to collection of the first partition in the generational algorithm, since we maintain a “remembered” list of all points from outside the car.
   - We identify objects with no references at all, as well as garbage cycles that are contained completely within this car.
   - reachable objects in the car are always moved to some other car, so each garbage-collected car becomes empty and can be removed from the train.

2. Sometimes, the first train has no external reference.
   - That is, there are no pointers from the root set to any car of the train, and the remembered sets for the cars contain only references from other cars in the train, not from other trains.
   - In this situation, the train is a huge collection of cyclic garbage, and we delete the entire train.
Generational GC works most frequently on the area of the heap that contains the youngest objects. It tends to collect a lot of garbage for relatively little work.

The train algorithm does not spend a large proportion of time on young objects. It does limit the pauses due to garbage collection. An advantage to the train algorithm is that we never have to do a complete garbage collection, as we do occasionally for generational garbage collection.

A good combination of strategies is to use generational collection for young objects, and once an becomes sufficiently mature, to “promote” it to a separate heap that is managed by the train algorithm.
Outline

1 Introduction
2 Data Storage
3 Stack management
4 Heap management
5 Garbage collection
6 Conclusion
■ **Run-time Organization**: To implement the abstractions embodied in the source language, a compiler creates and manages a run-time environment in concert with the operating system and the target machine. The run-time environment has static data areas for the object code and the static data objects created at compile time. It also has dynamic stack and heap areas for managing objects created and destroyed as the target program executes.

■ **Control Stack**: Procedure calls and returns are usually managed by a run-time stack called the control stack. We can use a stack because procedure calls or activations nest in time; that is, if p calls q, then this activation of q is nested within this activation of p.
Stack Allocation: Storage for local variables can be allocated on a run-time stack for languages that allow or require local variables to become inaccessible when their procedures end. For such languages, each live activation has an activation record (or frame) on the control stack, with the root of the activation tree at the bottom, and the entire sequence of activation records on the stack corresponding to the path in the activation tree to the activation where control currently resides. The latter activation has its record at the top of the stack.
Access to Nonlocal Data on the Stack: For languages like C that do not allow nested procedure declarations, the location for a variable is either global or found in the activation record on top of the run-time stack. For languages with nested procedures, we can access nonlocal data on the stack through access links, which are pointers added to each activation record. The desired nonlocal data is found by following a chain of access links to the appropriate activation record. A display is an auxiliary array, used in conjunction with access links, that provides an efficient short-cut alternative to a chain of access links.
**Heap Management:** The heap is the portion of the store that is used for data that can live indefinitely, or until the program deletes it explicitly. The memory manager allocates and deallocates space within the heap. Garbage collection finds spaces within the heap that are no longer in use and can therefore be reallocated to house other data items. For languages that require it, the garbage collector is an important subsystem of the memory manager.
**Exploiting Locality:** By making good use of the memory hierarchy, memory managers can influence the run time of a program. The time taken to access different parts of memory can vary from nanoseconds to milliseconds. Fortunately, most programs spend most of their time executing a relatively small fraction of the code and touching only a small fraction of the data. A program has **temporal locality** if it is likely to access the same memory locations again soon; it has **spatial locality** if it is likely to access nearby memory locations soon.
Reducing Fragmentation: As the program allocates and deallocates memory, the heap may get fragmented, or broken into large numbers of small noncontiguous free spaces or holes. The best fit strategy (allocate the smallest available hole that satisfies a request) has been found empirically to work well. While best fit tends to improve space utilization, it may not be best for spatial locality. Fragmentation can be reduced by combining or coalescing adjacent holes.

Manual Deallocation: Manual memory management has two common failings: not deleting data that can not be referenced is a memory-leak error, and referencing deleted data is a dangling-pointer-reference error.
Reachability: Garbage is data that cannot be referenced or reached. There are two basic ways of finding unreachable objects: either catch the transition as a reachable object turns unreachable, or periodically locate all the reachable objects and infer that all remaining objects are unreachable.

Reference-Counting Collectors: maintain a count of the references to an object; when the count transitions to zero, the object becomes unreachable. Such collectors introduce the overhead of maintaining references and can fail to find “cyclic” garbage, which consists of unreachable objects that reference each other, perhaps through a chain of references.
Trace-Based Garbage Collectors: iteratively examine or trace all references to find reachable objects, starting with the root set consisting of objects that can be accessed directly without having to dereference any pointers.

Mark-and-Sweep Collectors: visit and mark all reachable objects in a first tracing step and then sweep the heap to free up unreachable objects.

Mark-and-Compact Collectors: improve upon mark-and-sweep; they relocate objects in the heap to eliminate memory fragmentation.
**Key Concepts in the Chapter (#9)**

- **Copying Collectors**: break the dependency between tracing and finding free space. They partition the memory into two *semispaces*, A and B. Allocation requests are satisfied from one semispace, say A, until it fills up, at which point the garbage collector takes over, copies the reachable objects to the other space, say B, and reverses the roles of the semispaces.

- **Incremental Collectors**: Simple trace-based collectors stop the user program while garbage is collected. **Incremental collectors** interleave the actions of the garbage collector and the *mutator* or user program. The mutator can interfere with incremental reachability analysis, since it can change the references within previously scanned objects. Incremental collectors therefore play it safe by overestimating the set of reachable objects; any “floating garbage” can be picked up in the next round of collection.
Partial Collectors: also reduce pauses; they collect a subset of the garbage at a time. The best known of partial-collection algorithms, generational garbage collection, partitions objects according to how long they have been allocated and collects the newly created objects more often because they tend to have shorter lifetimes. An alternative algorithm, the train algorithm, uses fixed length partitions, called cars, that are collected into trains. Each collection step is applied to the first remaining car of the first remaining train. When a car is collected, reachable objects are moved out to the other cars, so this car is left with garbage and can be removed from the train. These two algorithms can be used together to create a partial collector that applies the generational algorithm to younger objects and the train algorithm to more mature objects.
The treadmill: real-time garbage collection without motion sickness.

A nonrecursive list compacting algorithm.

Recursive programming.

On-the-fly garbage collection: an exercise in cooperation.

A lisp garbage-collector for virtual memory computer systems.

Incremental collection of mature objects.

The C language calling sequence.
Bibliography of the Chapter (#2)

A real-time garbage collector based on the lifetimes of objects. 

Recursive functions of symbolic expressions and their computation by machine. 

ALGOL 60 Implementation. 
Academic Press.

Uniprocessor garbage collection techniques.
Compilation and Language Theory

2nd edition

Stéphane GALLAND
Module LO46

IRTES-SeT, UTBM
90010 Belfort cedex, France
stephane.galland@utbm.fr – http://www.multiagent.fr
Appendix
“Compilation and Language Theory” by Stéphane GALLAND is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License.

You are free

- To Share to copy, distribute and transmit the work
- To Remix to adapt the work

Under the following conditions

- Attribution You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).
- Noncommercial You may not use this work for commercial purposes.
- Share alike If you alter, transform, or build upon this work, you may distribute the resulting work only under the same or similar license to this one.

© 2012–2014 Stéphane Galland
stephane.galland@utbm.fr – http://www.multiagent.fr
Sources

The \LaTeX\ code of this document is available at https://bitbucket.org/sgalland/lo46-lessons.

Generation

This document is generated the January 16, 2014 with the following tools:

- \LaTeX.
- Beamer.
- IRTES style for beamer (http://www.multiagent.fr/SlideStyle).
- Auto\LaTeX (http://www.arakhne.org/autolatex).
Systems and Transport Laboratory
Institut de Recherche sur les Transports, l’Énergie et la Société
Université de Technologie de Belfort-Montbéliard, France

Topics: Multiagent systems, Agent-based simulation, Agent-oriented software engineering

Web page: http://www.multiagent.fr/People:Galland_stephane
Email: stephane.galland@utbm.fr

Open-source contributions:
- http://www.janus-project.org
- http://www.arakhne.org