Compilerconstructie

najaar 2012

`http://www.liacs.nl/home/rvvliet/coco/`

Rudy van Vliet
kamer 124 Snellius, tel. 071-527 5777
rvvliet(at)liacs.nl

college 1, dinsdag 4 september 2012

Overview

Why this course

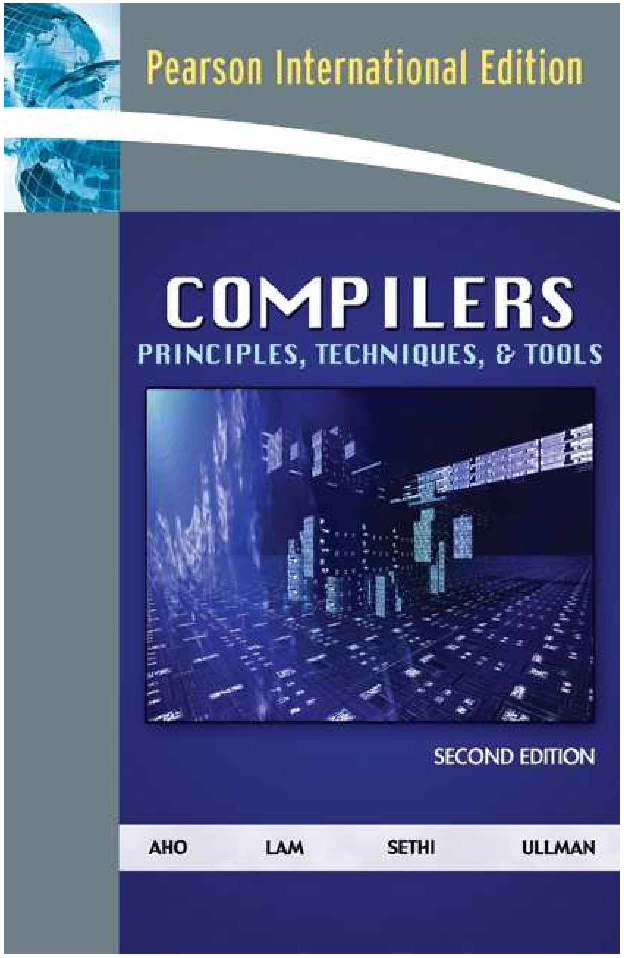
It's part of the general background of a software engineer

- How do compilers work?
- How do computers work?
- What machine code is generated for certain language constructs?
- Working on a non-trivial programming project

After the course

- Know how to build a compiler for a simplified progr. language
- Know how to use compiler construction tools, such as generators for scanners and parsers
- Be familiar with compiler analysis and optimization techniques

Course Outline



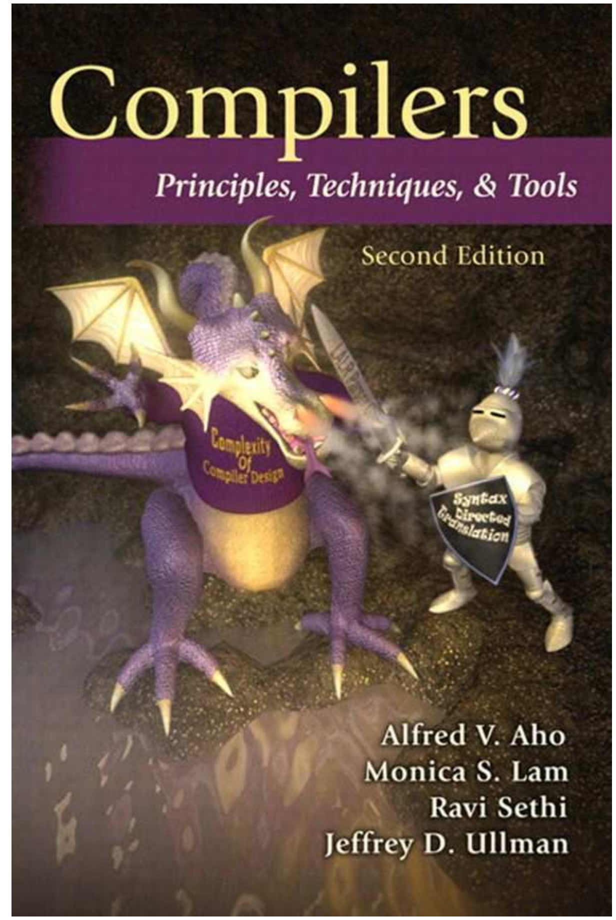
- In class, we discuss the theory using the ‘dragon book’ by Aho et al.
- The theory is applied in the practicum to build a compiler that converts Pascal code to MIPS instructions.

A.V. Aho, M.S. Lam, R. Sethi, and J.D. Ullman,

Compilers: Principles, Techniques, & Tools,

Addison-Wesley, 2007, ISBN: 978-0-321-49169-5 (international edition).

Course Outline

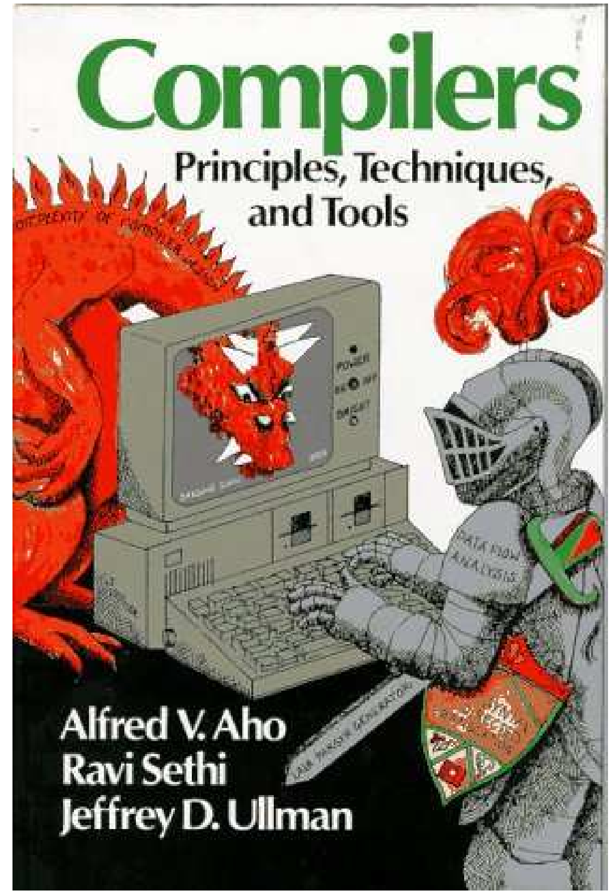


- In class, we discuss the theory using the 'dragon book' by Aho et al.
- The theory is applied in the practicum to build a compiler that converts Pascal code to MIPS instructions.

A.V. Aho, M.S. Lam, R. Sethi, and J.D. Ullman,
Compilers: Principles, Techniques, & Tools,
Addison-Wesley, 2006, ISBN: 978-0-321-54798-9.

Earlier edition

- Dragon book has been revised in 2006
- In Second edition good improvements are made
  - Parallelism
    - ...
    - Array data-dependence analysis
- First edition may also be used



A.V. Aho, R. Sethi, and J.D. Ullman,
Compilers: Principles, Techniques, and Tools,
Addison-Wesley, 1986, ISBN-10: 0-201-10088-6 / 0-201-10194-7 (international edition).

Course Outline

- Contact
  - Room 124, tel. 071-5275777, rvvliet(at)liacs.nl
  - Course website: http://www.liacs.nl/home/rvvliet/coco
    Lecture slides, assignments, grades

- Practicum
  - 4 self-contained assignments
  - These assignments are done by groups of two persons
  - Assignments are submitted by e-mail
  - Assistants: Teddy Zhai, Sven van Haastregt

Course Outline

- Grading:
  Combination of the grades from the written exam and the practicum

- You need to pass all 4 assignments to obtain a grade

- Final grade is only accepted if all grades are ≥ 5.5

- Then, you obtain 6 EC

Studying only from the lecture slides may not be sufficient.
Relevant book chapters will be given.

Course Outline

(tentative)

1. Overview
2. Lexical Analysis
3. Syntax Analysis Part 1
4. Syntax Analysis Part 2
5. Assignment 1
6. Static Type Checking
7. Assignment 2
8. Intermediate Code Generation
9. Assignment 3
10. Code Generation
11. Code optimization
12. Assignment 4
13. Daedalus

Practicum

- Assignment 1: Calculator
- Assignment 2: Parsing & Syntax tree
- Assignment 3: Intermediate code
- Assignment 4: Assembly generation

2 academic hours of Lab session + 3 weeks to complete (except assignment 1)

Short History of Compiler Construction

Formerly 'a mystery', today one of the best known areas of computing

1957 Fortran first compilers
(arithmetic expressions, statements, procedures)

1960 Algol first formal language definition
(grammars in Backus-Naur form, block structure, recursion, ...)

1970 Pascal user-defined types, virtual machines (P-code)

1985 C++ object-orientation, exceptions, templates

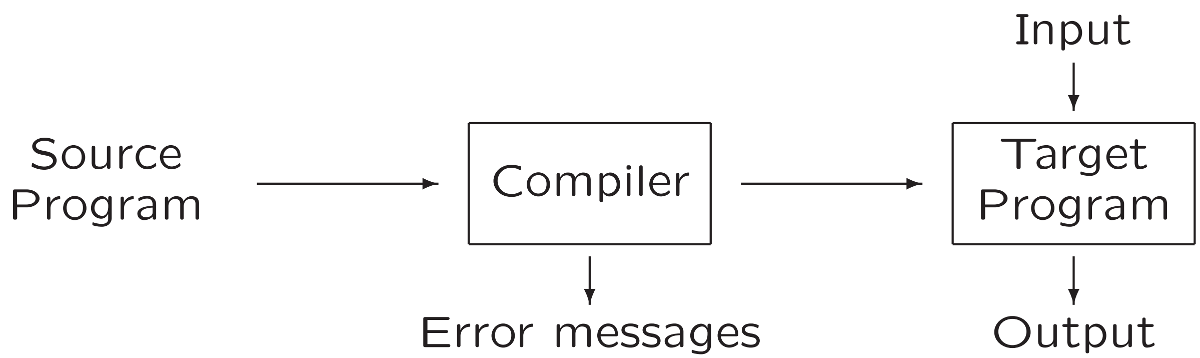
1995 Java just-in-time compilation

We only consider imperative languages

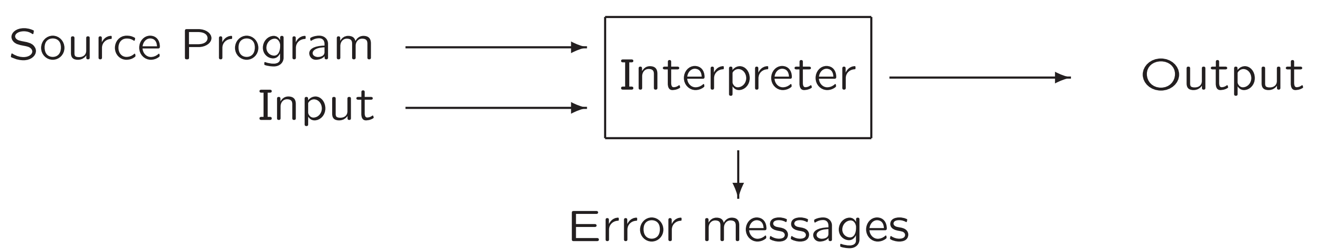
Functional languages (e.g., Lisp) and logical languages (e.g., Prolog) require different techniques.

Compilers and Interpreters

- Compilation:
  Translation of a program written in a source language into a semantically equivalent program written in a target language



- Interpretation:
  Performing the operations implied by the source program.

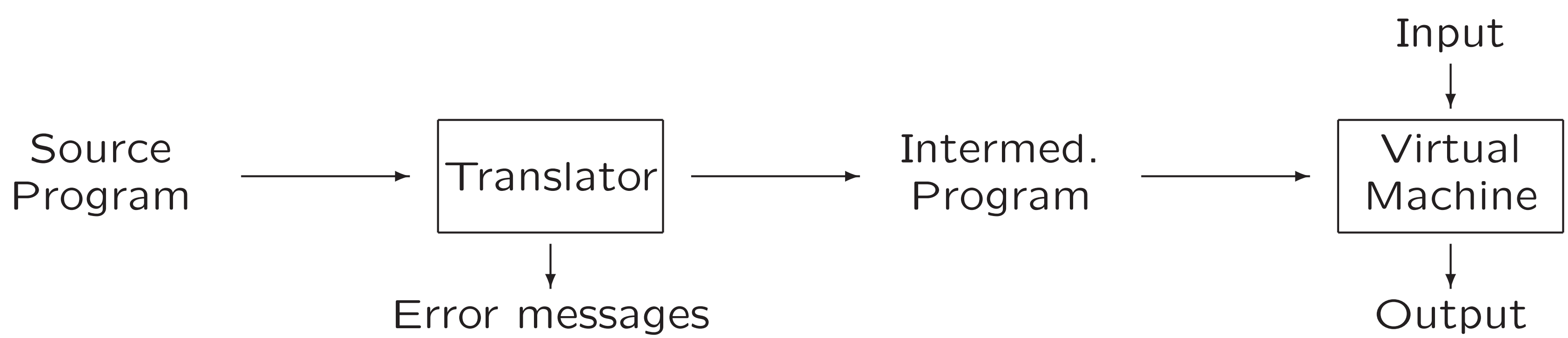


Compilers and Interpreters

- Compiler: Translates source code into machine code, with scanner, parser, . . . , code generator
- Interpreter: Executes source code 'directly', with scanner, parser
  Statements in, e.g., a loop are scanned and parsed again and again

Compilers and Interpreters

- Hybrid compiler (Java):
  - Translation of a program written in a source language into a semantically equivalent program written in an intermediate language (bytecode)
  - Interpretation of intermediate program by virtual machine, which simulates physical machine



Analysis-Synthesis Model of Compilation

There are two parts to compilation:

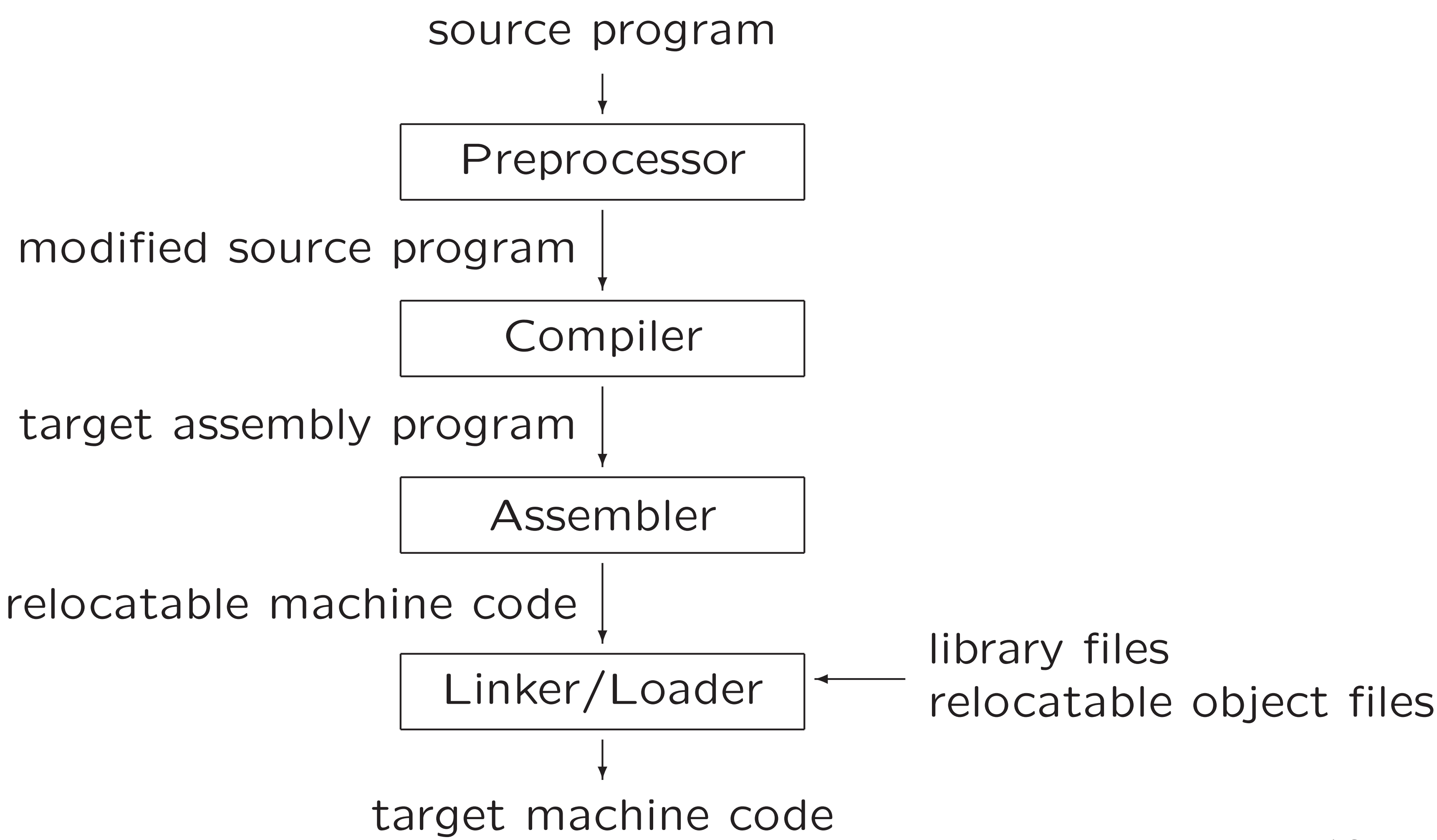
- Analysis
  - Determines the operations implied by the source program which are recorded in an intermediate representation, e.g., a tree structure

- Synthesis
  - Takes the intermediate representation and translates the operations therein into the target program

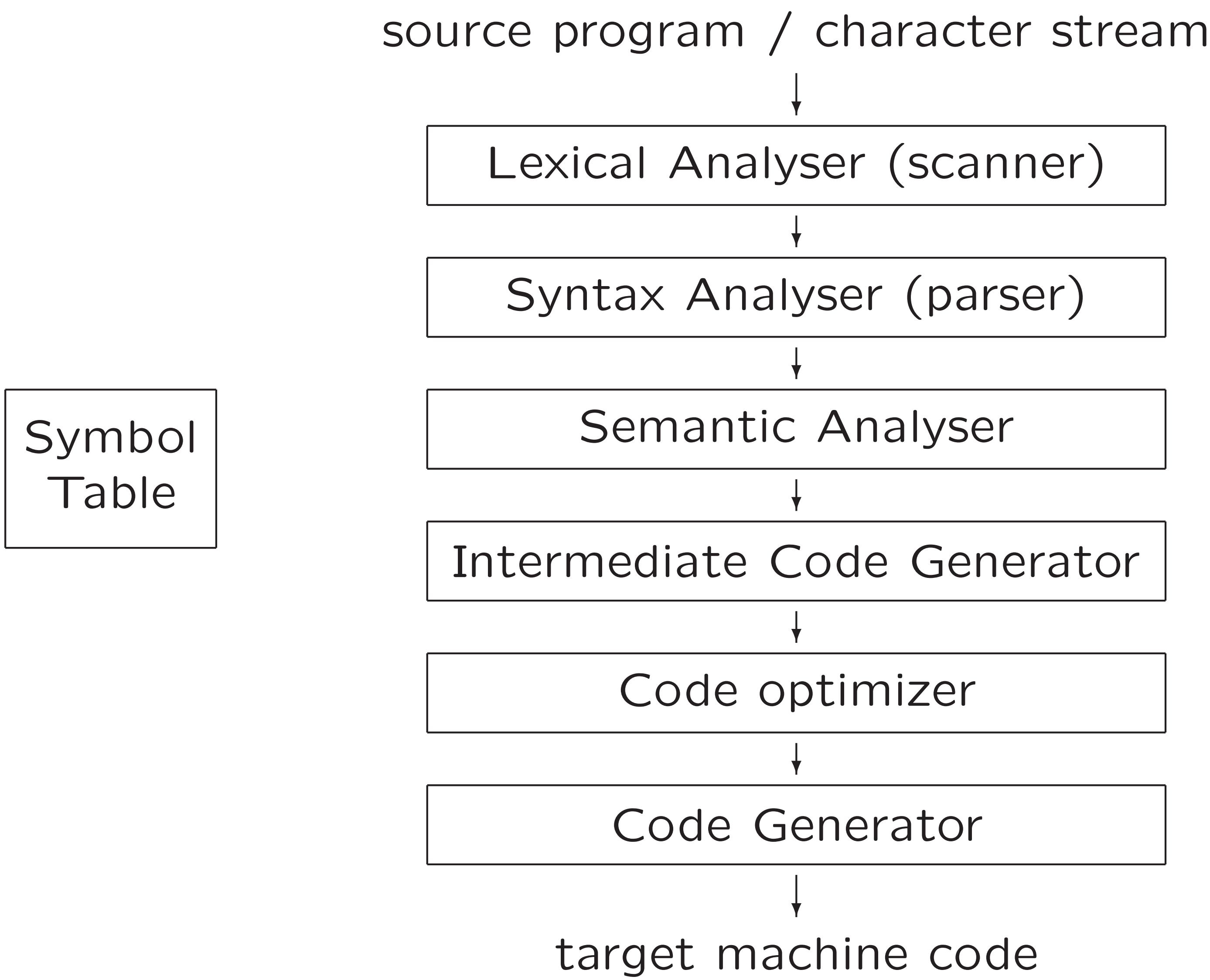
Other tools that use A-S Model

- Editors (syntax highlighting, text auto completion)
- Text formatters (LaTeX, MS Word)

Compilation flow



The Phases of a Compiler



The Phases of a Compiler

Character stream:

```
position = intitial + rate * 60
```

Lexical Analyser (scanner)

Token stream:

$$\langle \mathbf{id}, 1\rangle \ \langle = \rangle \ \langle \mathbf{id}, 2\rangle \ \langle + \rangle \ \langle \mathbf{id}, 3\rangle \ \langle * \rangle \ \langle 60 \rangle$$

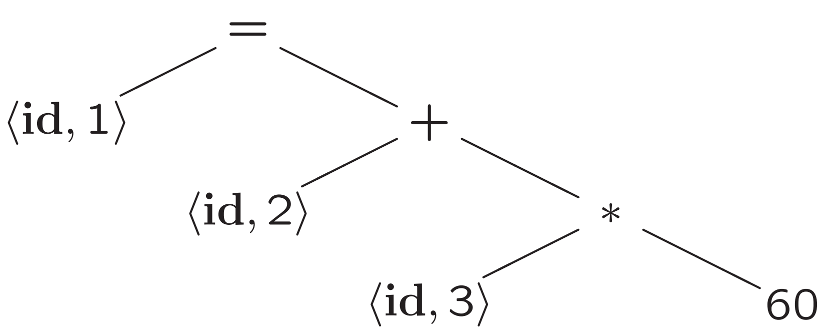
The Phases of a Compiler

Token stream:

$$\langle \mathbf{id}, 1\rangle \ \langle = \rangle \ \langle \mathbf{id}, 2\rangle \ \langle + \rangle \ \langle \mathbf{id}, 3\rangle \ \langle * \rangle \ \langle 60 \rangle$$

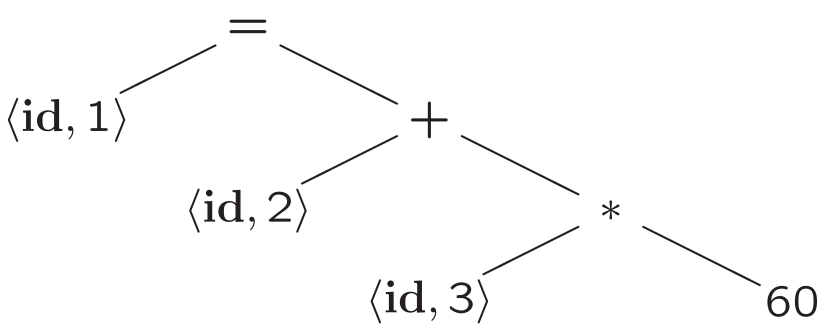
Syntax Analyser (parser)

Parse/syntax tree

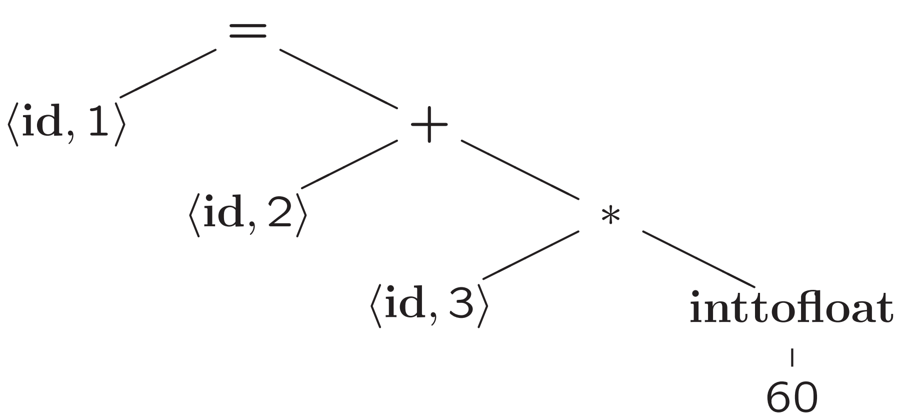


The Phases of a Compiler

Parse/syntax tree

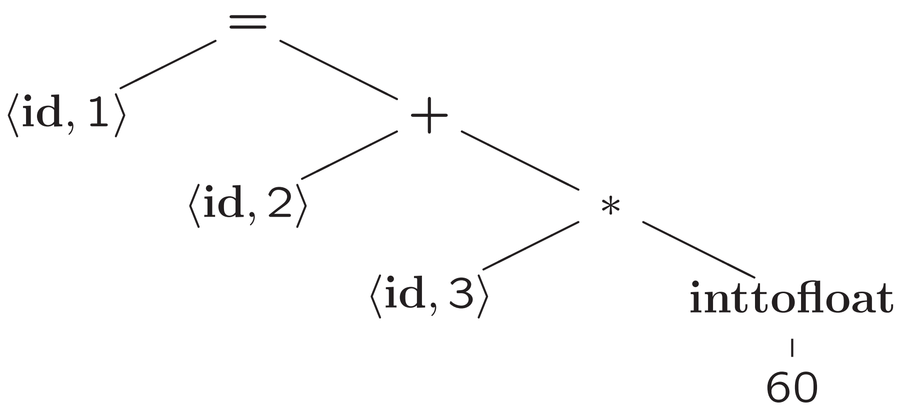


Semantic Analyser (parser)



The Phases of a Compiler

Parse/syntax tree



Intermediate Code Generator

Intermediate code (three-address code):

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

The Phases of a Compiler

Intermediate code (three-address code):

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

Code Optimizer

```
t1 = id3 * 60.0
id1 = id2 + t1
```

The Phases of a Compiler

```
t1 = id3 * 60.0
id1 = id2 + t1
```

Code Generator

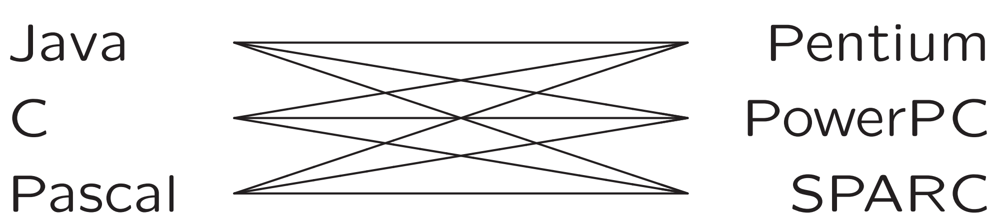
```
LDF  R2, id3
MULF R2, R2, #60.0
LDF  R1, id2
ADDF R1, R1, R2
STF  id1, R1
```

The Grouping of Phases

- Front End:
  scanning, parsing, semantical analysis
  (source code $\rightarrow$ intermediate representation)

- Back End:
  code optimizing, code generation
  (intermediate representation $\rightarrow$ target machine code)

language-dependent **machine-dependent**



Passes: Single-Pass Compilers

Phases work in an interleaved way

```
do
  scan token
  parse token
  check token
  generate code for token
while (not eof)
```

Portion of code is generated while reading portion of source program

Passes: Multi-Pass Compilers

Phases are separate 'programs', which run sequentially

characters $\longrightarrow$ Scanner $\longrightarrow$ tokens $\longrightarrow$ Parser $\longrightarrow$ tree
$\longrightarrow$ Semantical analyser $\longrightarrow \ldots \longrightarrow$ code

Each phase reads from a file and writes to a new file.

Time vs memory

Why multi-pass?

- If the language is complex
- If portability is important

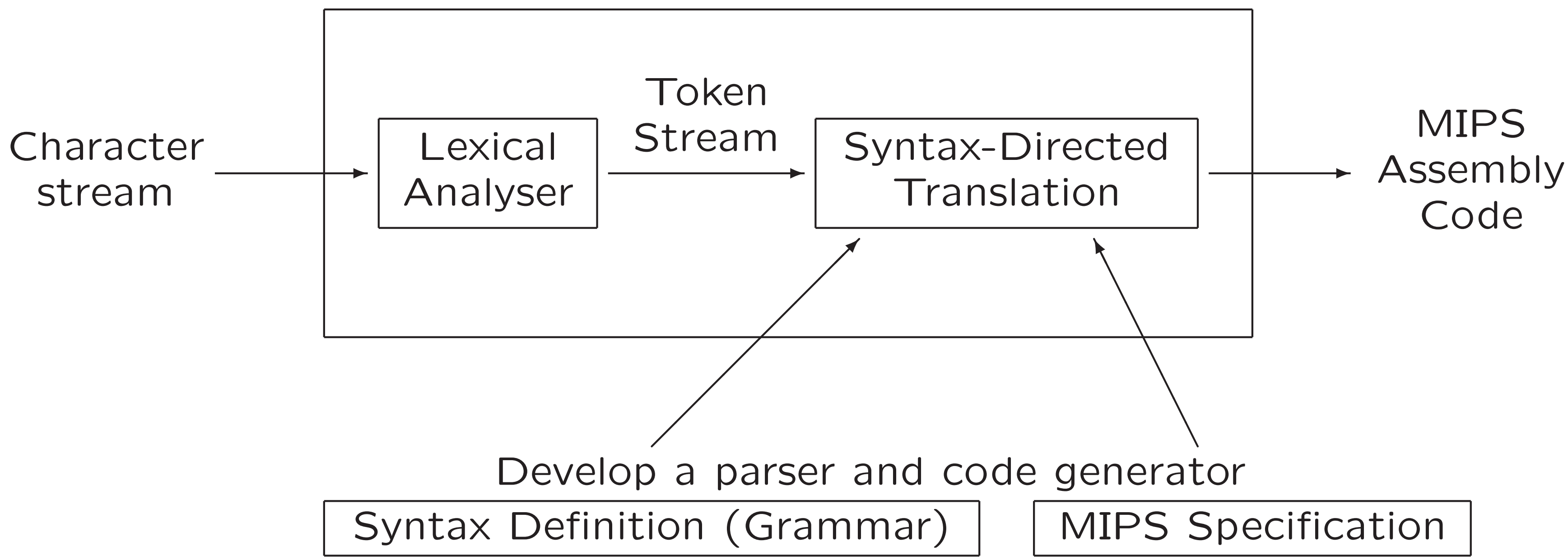
Today: often two-pass compiler

Compiler-Construction Tools

Software development tools are available to implement one or more compiler phases

- Scanner generators
- Parser generators
- Syntax-directed translation engines
- Automatic code generators
- Data-flow engines

The Structure of our compiler



Syntax directed translation:
The compiler uses the syntactic structure of the language to generate output

What is a grammar?

Context-free grammar is a 4-tuple with

- A set of tokens (*terminal* symbols)
- A set of *nonterminals* (syntactic variables)
- A set of *productions*: rules how to decompose nonterminals
- A designated *start/* symbol (nonterminal)

Example: Context-free grammar for simple expressions:

$$G = (\{\textit{list}, \textit{digit}\},\ \{+, -, \mathbf{0}, \mathbf{1}, \mathbf{2}, \mathbf{3}, \mathbf{4}, \mathbf{5}, \mathbf{6}, \mathbf{7}, \mathbf{8}, \mathbf{9}\},\ P, \textit{list})$$

with productions P:

$$\begin{array}{rcl} \textit{list} & \rightarrow & \textit{list} + \textit{digit} \\ \textit{list} & \rightarrow & \textit{list} - \textit{digit} \\ \textit{list} & \rightarrow & \textit{digit} \\ \textit{digit} & \rightarrow & \mathbf{0} \mid \mathbf{1} \mid \mathbf{2} \mid \mathbf{3} \mid \mathbf{4} \mid \mathbf{5} \mid \mathbf{6} \mid \mathbf{7} \mid \mathbf{8} \mid \mathbf{9} \end{array}$$

Derivation

Given a context-free grammar, we can determine the set of all strings (sequences of tokens) generated by the grammar using derivations:

- We begin with the start symbol

- In each step, we replace one nonterminal in the current form with one of the right-hand sides of a production for that nonterminal

Derivation (Example)

$$G = (\{\mathit{list}, \mathit{digit}\},\ \{+, -, \mathbf{0}, \mathbf{1}, \mathbf{2}, \mathbf{3}, \mathbf{4}, \mathbf{5}, \mathbf{6}, \mathbf{7}, \mathbf{8}, \mathbf{9}\},\ P, \mathit{list})$$
$$\mathit{list} \rightarrow \mathit{list} + \mathit{digit} \mid \mathit{list} - \mathit{digit} \mid \mathit{digit}$$
$$\mathit{digit} \rightarrow \mathbf{0} \mid \mathbf{1} \mid \mathbf{2} \mid \mathbf{3} \mid \mathbf{4} \mid \mathbf{5} \mid \mathbf{6} \mid \mathbf{7} \mid \mathbf{8} \mid \mathbf{9}$$

Example: 9-5+2

$$\begin{aligned}
\underline{\mathit{list}} &\Rightarrow \underline{\mathit{list}} + \mathit{digit} \\
&\Rightarrow \underline{\mathit{list}} - \mathit{digit} + \mathit{digit} \\
&\Rightarrow \underline{\mathit{digit}} - \mathit{digit} + \mathit{digit} \\
&\Rightarrow 9 - \underline{\mathit{digit}} + \mathit{digit} \\
&\Rightarrow 9 - 5 + \underline{\mathit{digit}} \\
&\Rightarrow 9 - 5 + 2
\end{aligned}$$

This is an example of leftmost derivation, because we replaced the leftmost nonterminal (underlined) in each step

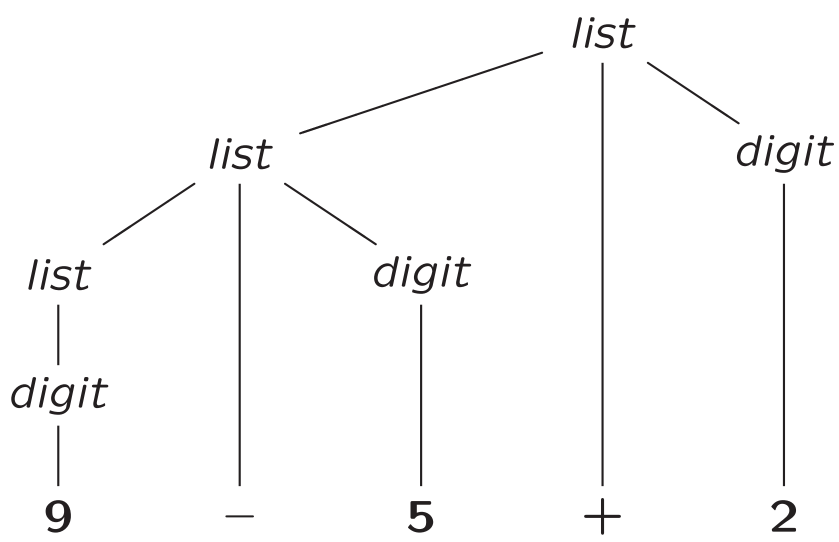
Parse Tree

(derivation tree in FI2)

- The root of the tree is labelled by the start symbol
- Each leaf of the tree is labelled by a terminal (=token) or ϵ (=empty)
- Each interior node is labelled by a nonterminal
- If node A has children $X_1, X_2, \ldots, X_n$, then there must be a production $A \rightarrow X_1 X_2 \ldots X_n$

Parse Tree (Example)

Parse tree of the string $\mathbf{9} - \mathbf{5} + \mathbf{2}$ using grammar G



Yield of the parse tree: the sequence of leafs (left to right)

Parsing: the process of finding a parse tree for a given string

Language: the set of strings that can be generated by some parse tree

Ambiguity

Consider the following context-free grammar:

$$G' = (\{\textit{string}\},\ \{+, -, \mathbf{0}, \mathbf{1}, \mathbf{2}, \mathbf{3}, \mathbf{4}, \mathbf{5}, \mathbf{6}, \mathbf{7}, \mathbf{8}, \mathbf{9}\},\ P, \textit{string})$$

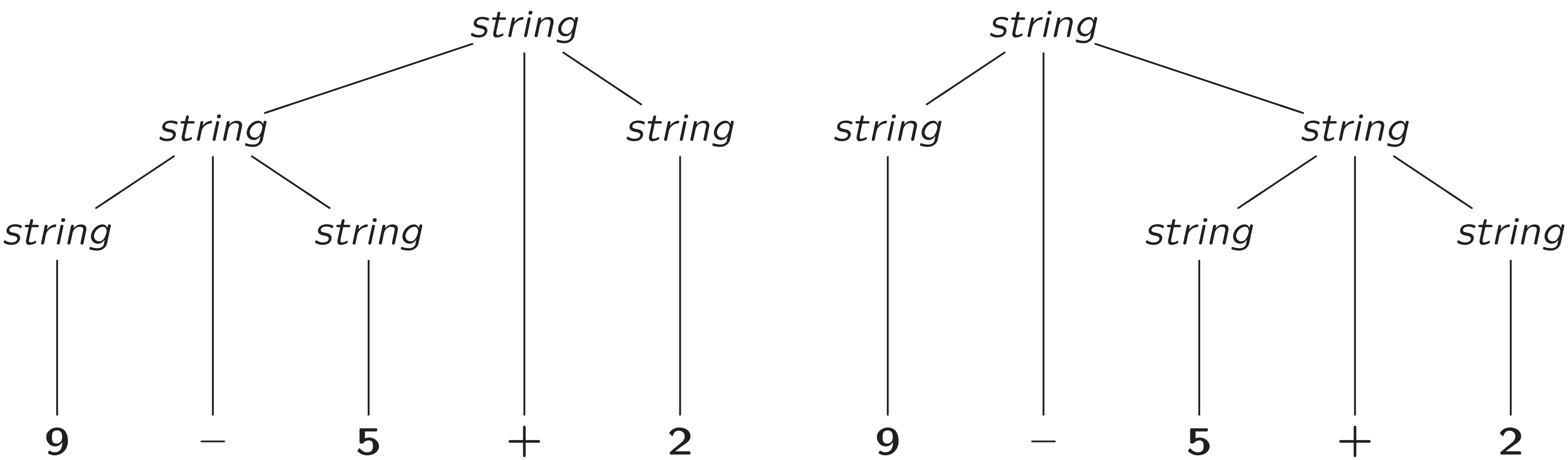
with productions P

$$\textit{string} \rightarrow \textit{string} + \textit{string} \mid \textit{string} - \textit{string} \mid \mathbf{0} \mid \mathbf{1} \mid \ldots \mid \mathbf{9}$$

This grammar is ambiguous, because more than one parse tree generates the string $\mathbf{9} - \mathbf{5} + \mathbf{2}$

Ambiguity (Example)

Parse trees of the string $\mathbf{9-5+2}$ using grammar G'



$(9-5)+2=6$

$9-(5+2)=2$

Associativity of Operators

By convention

$$\left.\begin{array}{rcl} 9+5+2 & = & (9+5)+2 \\ 9-5-2 & = & (9-5)-2 \end{array}\right\} \text{left associative}$$

In most programming languages:

$+, -, *, /$ are left associative

$**, =$ are right associative:

$$\begin{array}{rcl} a**b**c & = & a**(b**c) \\ a=b=c & = & a=(b=c) \end{array}$$

Precedence of Operators

Consider: $9 + 5 * 2$
Is this $(9 + 5) * 2$ or $9 + (5 * 2)$?
Associativity does not resolve this

Precedence of operators: $+ \; -$ / $* \; /$ ↓ increasing precedence

A grammar for arithmetic expressions:

$$
\begin{array}{rcl}
expr & \rightarrow & expr + term \mid expr - term \mid term \\
term & \rightarrow & term * factor \mid term / factor \mid factor \\
factor & \rightarrow & digit \mid (expr) \\
digit & \rightarrow & \mathbf{0} \mid \mathbf{1} \mid \ldots \mid \mathbf{9}
\end{array}
$$

Parse tree for $\mathbf{9} + \mathbf{5} * \mathbf{2}$ . . .

Syntax-Directed Translation

Using the syntactic structure of the language to generate output corresponding to some input

Two techniques:

- Syntax-directed definition
- Translation scheme

Example: translation of infix notation to postfix notation

infix	postfix
$(9-5)+2$	$95-2+$
$9-(5+2)$	$952+-$

What is $952+-3*$?

Syntax-Directed Translation

Using the syntactic structure of the language to generate output corresponding to some input

Two variants:

- Syntax-directed definition
- Translation scheme

Example: translation of infix notation to postfix notation

Simple infix expressions generated by

$$\begin{array}{lcl} \mathit{expr} & \rightarrow & \mathit{expr}_1 + \mathit{term} \mid \mathit{expr}_1 - \mathit{term} \mid \mathit{term} \\ \mathit{term} & \rightarrow & 0 \mid 1 \mid \ldots \mid 9 \end{array}$$

Syntax-Directed Definition

- Uses a context-free grammar to specify the syntactic structure of the language

- Associates a set of *attributes* with (non)terminals

- Associates with each production a set of *semantic rules* for computing values for the attributes

- The attributes contain the translated form of the input after the computations are completed
(in example: postfix notation corresponding to subtree)

Syntax-Directed Definition (Example)

Production	Semantic rule
$expr \rightarrow expr_1 + term$	$expr.t = expr_1.t \;\|\|\; term.t \;\|\|\; \text{'}+\text{'}$
$expr \rightarrow expr_1 - term$	$expr.t = expr_1.t \;\|\|\; term.t \;\|\|\; \text{'}-\text{'}$
$expr \rightarrow term$	$expr.t = term.t$
$term \rightarrow 0$	$term.t = \text{'}0\text{'}$
$term \rightarrow 1$	$term.t = \text{'}1\text{'}$
...	...
$term \rightarrow 9$	$term.t = \text{'}9\text{'}$

Result: annotated parse tree

Example: $9 - 5 + 2$

Synthesized and Inherited Attributes

An attribute is said to be . . .

- synthesized if its value at a parse tree node N is determined from attribute values at the children of N (and at N itself)

- inherited if its value at a parse tree node N is determined from attribute values at the parent of N (and at N itself and its siblings)

We consider synthesized attributes

Depth-First Traversal

- A syntax-directed definition does not impose an evaluation order of the attributes on a parse tree

- Different orders might be suitable

- *Tree traversal*: a specific order to visit the nodes of a tree (always starting from the root node)

- Depth-first traversal
  - Start from root
  - Recursively visit children (in any order)
  - Hence, visit nodes far away from the root as quickly as it can (DF)

A Possible DF Traversal

Postorder traversal

```
procedure visit (node N)
{
  for (each child C of N, from left to right)
  { visit (C);
  }
  evaluate semantic rules at node N;
}
```

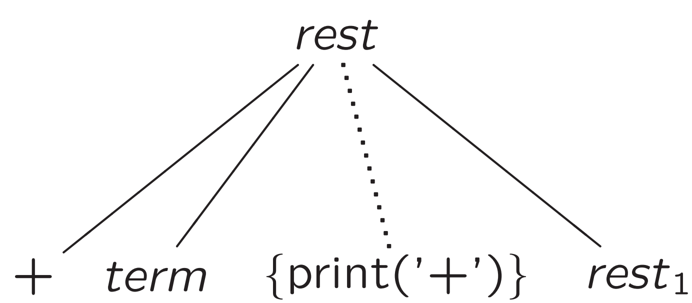
Can be used to determine synthesized attributes / annotated parse tree

Translation Scheme

A translation scheme is a context-free grammar
with semantic actions embedded in the bodies of the productions

Example

$$\textit{rest} \rightarrow +\textit{term}\ \{\text{print('+')}\}\ \textit{rest}_1$$



Translation Scheme (Example)

$$
\begin{array}{rcl}
\textit{expr} & \rightarrow & \textit{expr}_1 + \textit{term}\ \{\text{print('+')}\} \\
\textit{expr} & \rightarrow & \textit{expr}_1 - \textit{term}\ \{\text{print('-')}\} \\
\textit{expr} & \rightarrow & \textit{term} \\
\textit{term} & \rightarrow & 0\ \{\text{print('0')}\} \\
\textit{term} & \rightarrow & 1\ \{\text{print('1')}\} \\
\ldots & & \ldots \\
\textit{term} & \rightarrow & 9\ \{\text{print('9')}\}
\end{array}
$$

Example: parse tree for $9 - 5 + 2$

Implementation requires postorder traversal

Parsing

- Process of determining if a string of tokens can be generated by a grammar
- For any context-free grammar, there is a parser that takes at most $\mathcal{O}(n^3)$ time to parse a string of n tokens
- Linear algorithms sufficient for parsing programming languages
- Two methods of parsing:
  - Top-down constructs parse tree from root to leaves
  - Bottom-up constructs parse tree from leaves to root

Cf. top-down PDA and bottom-up PDA in FI2

Parsing (Top-Down Example)

$$
\begin{array}{rcl}
\textit{stmt} & \rightarrow & \textbf{expr}\ ; \\
 & | & \textbf{if}\ (\textbf{expr}\ )\textit{stmt} \\
 & | & \textbf{for}\ (\textit{optexpr}\ ; \textit{optexpr}\ ; \textit{optexpr}\ )\textit{stmt} \\
 & | & \textbf{other} \\
\textit{optexpr} & \rightarrow & \epsilon \\
 & | & \textbf{expr}
\end{array}
$$

How to determine parse tree for

$$\textbf{for}\ (;\textbf{expr}\ ;\textbf{expr}\ )\textbf{other}$$

Use lookahead: current terminal in input

Predictive Parsing

- Recursive-descent parsing is a top-down parsing method:
  - Executes a set of recursive procedures to process the input
  - Every nonterminal has one (recursive) procedure parsing the nonterminal's syntactic category of input tokens

- Predictive parsing is a special form of recursive-descent parsing:
  - The lookahead symbol unambiguously determines the production for each nonterminal

Predictive Parsing (Example)

```
void stmt()
{ switch (lookahead)
  { case expr:
           match(expr); match(';'); break;
    case if:
           match(if); match('('); match(expr); match(')'); stmt();
           break;
    case for:
           match(for); match('(');
           optexpr(); match(';'); optexpr(); match(';'); optexpr();
           match(')'); stmt(); break;
    case other;
           match(other); break;
    default:
           report("syntax error");
  }
}

void match(terminal t)
{ if (lookahead==t) lookahead = nextTerminal;
  else report("syntax error");
}
```

Using FIRST

- Let α be string of grammar symbols
- FIRST(α) is the set of terminals that appear as first symbols of strings generated from α

Simple example:

$$
\begin{array}{lcl}
\textit{stmt} & \rightarrow & \textbf{expr}\ ; \\
 & | & \textbf{if}\ (\textbf{expr}\ )\textit{stmt} \\
 & | & \textbf{for}\ (\textit{optexpr}\ ;\textit{optexpr}\ ;\textit{optexpr}\ )\textit{stmt} \\
 & | & \textbf{other}
\end{array}
$$

Right-hand side may start with nonterminal. . .

Using FIRST

- Let α be string of grammar symbols
- FIRST(α) is the set of terminals that appear as first symbols of strings generated from α
- When a nontermimal has multiple productions, e.g.,

$$A \rightarrow \alpha \mid \beta$$

then FIRST(α) and FIRST(β) must be disjoint in order for predictive parsing to work

Compiler constructie

college 1
Overview

Chapters for reading: 1.1, 1.2, 2.1-2.5