

# Compilerconstructie

najaar 2016

<http://www.liacs.leidenuniv.nl/~vlietrvan1/coco/>

**Rudy van Vliet**

kamer 124 Snellius, tel. 071-527 5777

rvvliet(at)liacs(dot)nl

college 1, woensdag 7 september 2016

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

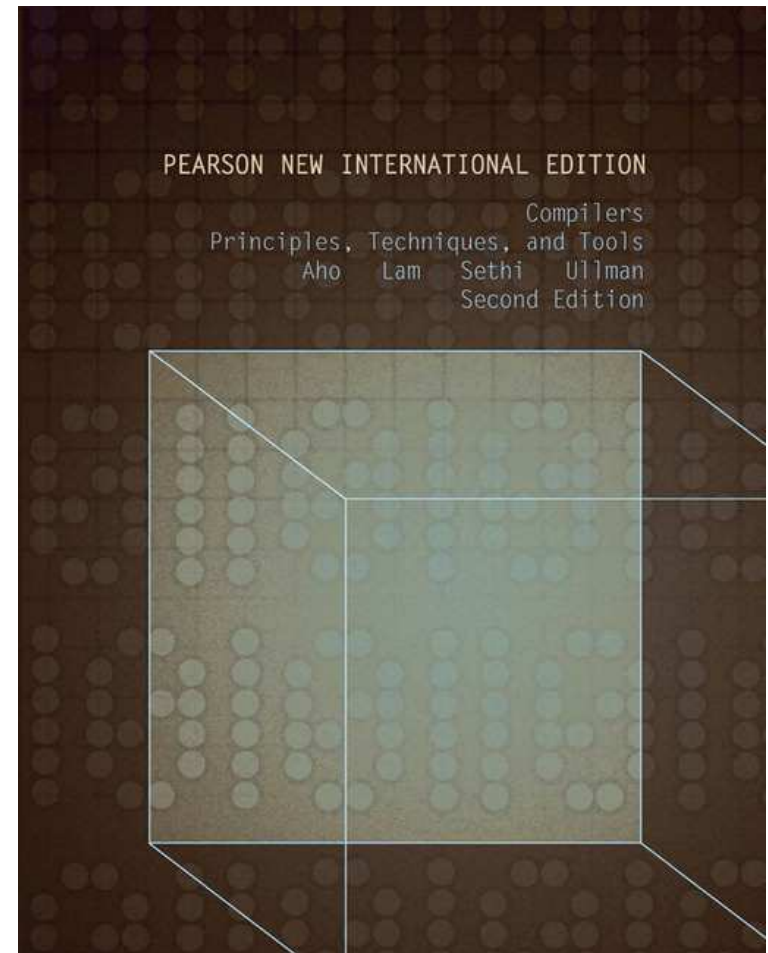
- Know how to build a compiler for a simplified progr. language
- Know how to use compiler construction tools, such as generators for scanners and parsers
- Be familiar with compiler analysis and optimization techniques

# Prior Knowledge

- Algoritmiek
- Fundamentele Informatica 2

# Course Outline

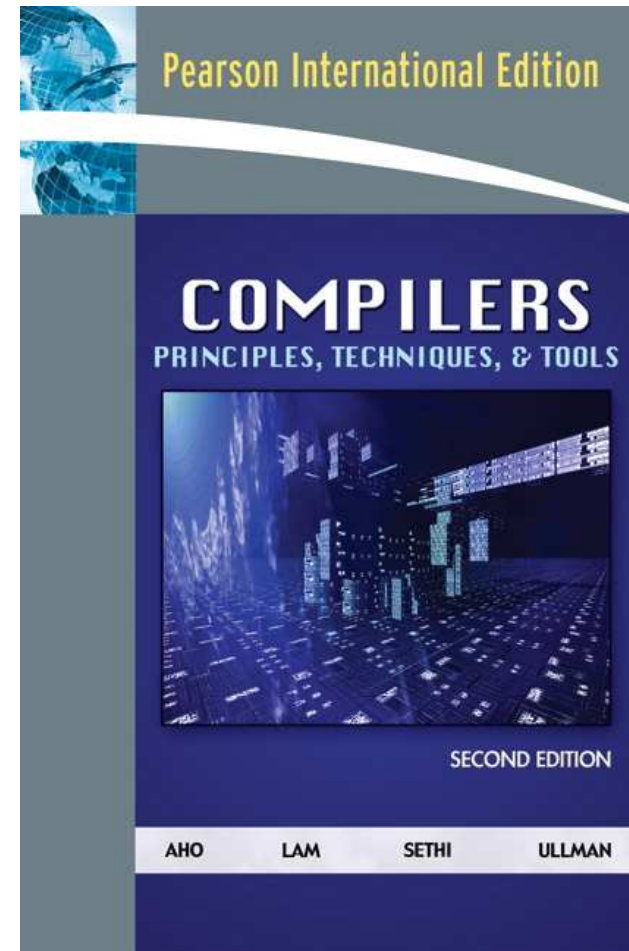
- In class, we discuss the theory using the 'dragon book' by Aho et al.
- The theory is applied in the practicum to build a compiler that converts Pascal code to MIPS instructions.



A.V. Aho, M.S. Lam, R. Sethi, and J.D. Ullman,  
*Compilers: Principles, Techniques, and Tools* (second edition),  
Pearson, 2013, ISBN: 978-1-29202-434-9 (international edition).

# Course Outline

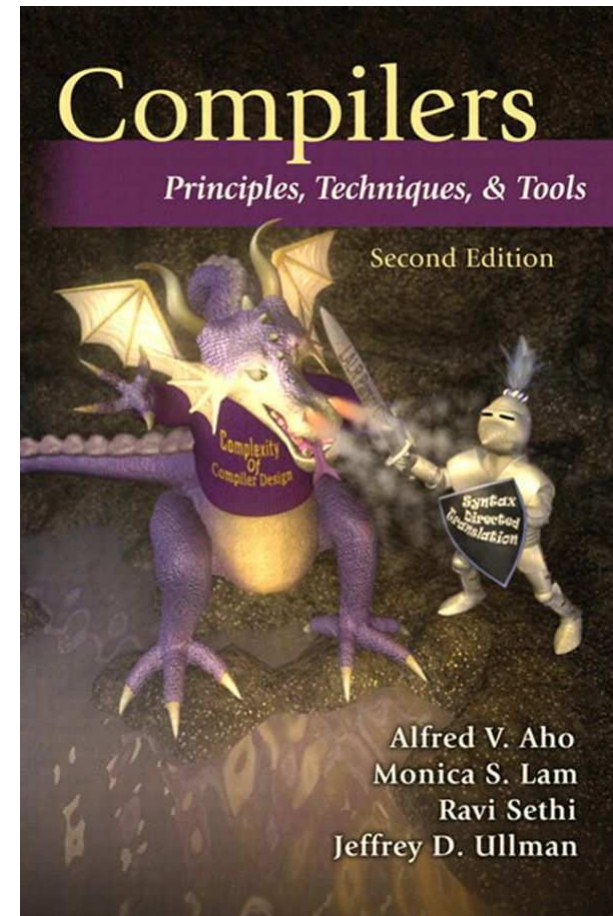
- In class, we discuss the theory using the 'dragon book' by Aho et al.
- The theory is applied in the practicum to build a compiler that converts Pascal code to MIPS instructions.



A.V. Aho, M.S. Lam, R. Sethi, and J.D. Ullman,  
*Compilers: Principles, Techniques, & Tools* (second edition),  
Pearson, 2007, ISBN: 978-0-321-49169-5 (international edition).

# Course Outline

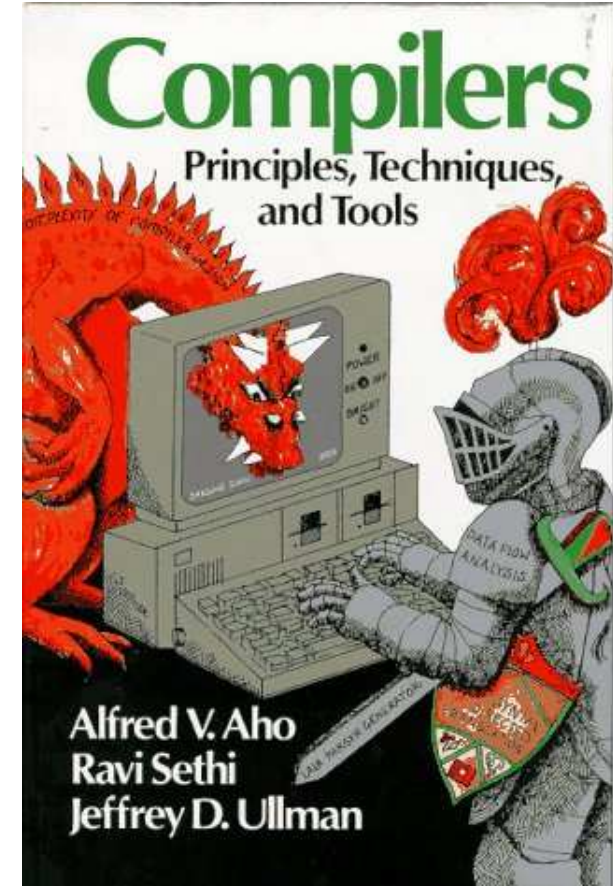
- In class, we discuss the theory using the 'dragon book' by Aho et al.
- The theory is applied in the practicum to build a compiler that converts Pascal code to MIPS instructions.



A.V. Aho, M.S. Lam, R. Sethi, and J.D. Ullman,  
*Compilers: Principles, Techniques, & Tools* (second edition),  
Pearson, 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, but not recommended



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\(dot\)nl](mailto:rvvliet@liacs.nl)
  - Course website:  
<http://www.liacs.leidenuniv.nl/~vlietrvan1/coco/>  
Lecture slides, assignments, grades
- Practicum
  - 4 self-contained assignments
  - Teams of **two** students
  - Assignments are submitted by e-mail
  - Assistant: Dennis Roos
- Written exam
  - 19 December 2016, 14:00–17:00
  - 7 March 2017, 14:00–17:00



# Course Outline

- You need to pass all 4 assignments and the written exam to obtain a sufficient grade
- Then, you obtain 6 EC
- Algorithm to compute final grade:

```
if (E >= 5.5)
{ if (A2,A3,A4 >= 5.5)
  { P = (A2+A3+A4)/3;
    F = (E+P)/2;
  }
  else
    F is undefined;
}
else
  F = E;
```

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

# Course Outline

(tentative)

1. Overview
2. Symbol Table / Lexical Analysis
3. Syntax Analysis 1 (+ exercise class)
4. Syntax Analysis 2 (+ exercise class)
5. Assignment 1
6. Static Type Checking
7. Assignment 2
8. Intermediate Code Generation 1 (+ lab session Friday)
9. Intermediate Code Generation 2 (+ exercise class?)
10. Assignment 3
11. Storage Organization and Code Generation  
(+ exercise class + lab session Friday)
12. Code optimization 1 (+ exercise class)
13. Assignment 4
14. Code Optimization 2 (+ exercise class + lab session Friday)

# Practicum

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

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

**Strict deadlines** (with one second chance)

# 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

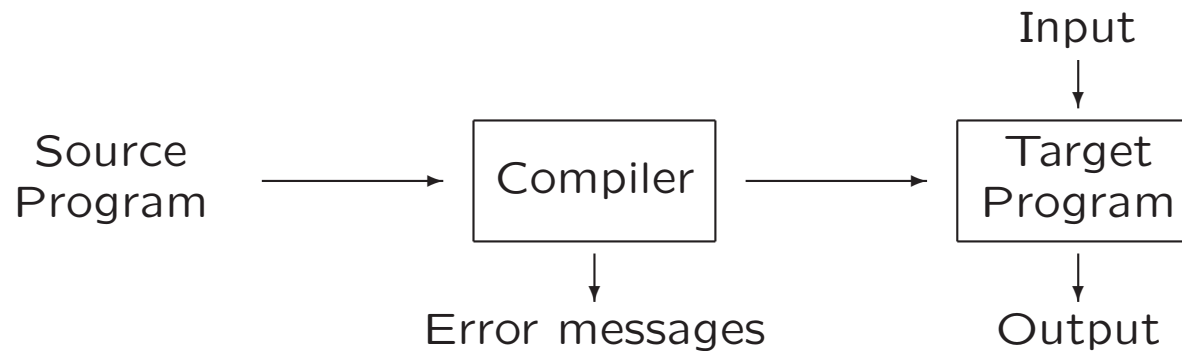
We only consider **imperative languages**

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

# 1.1 Language Processors

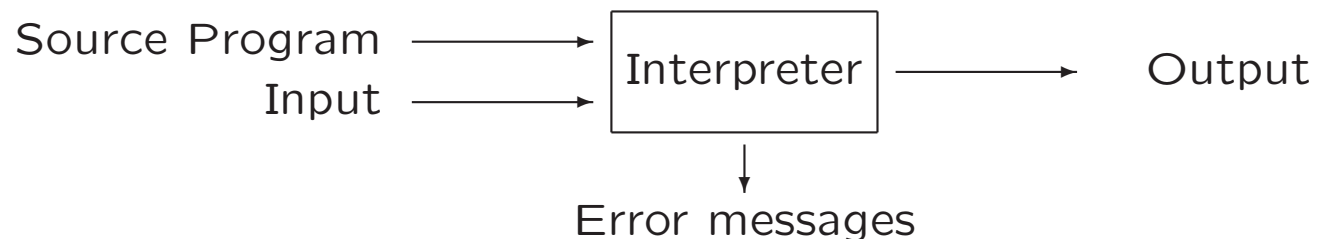
- Compilation:

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.

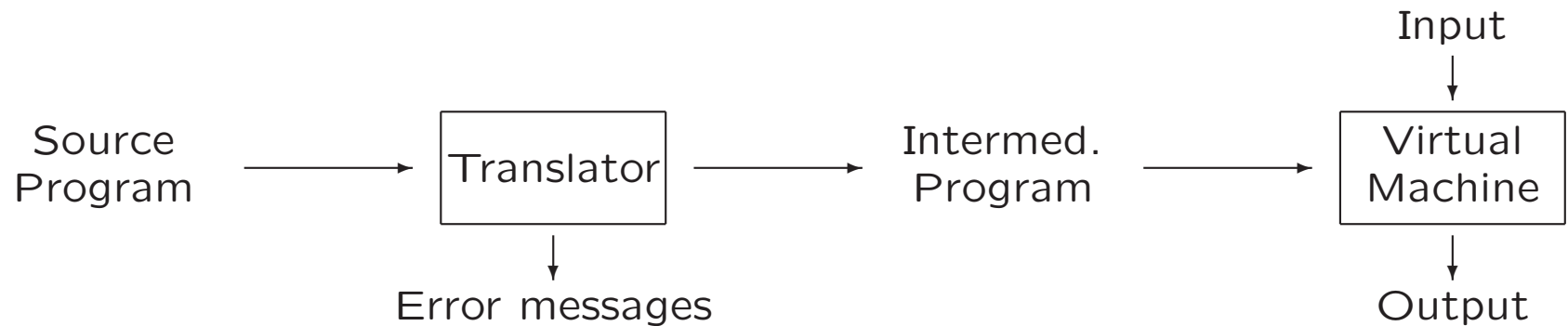


# Compilers and Interpreters

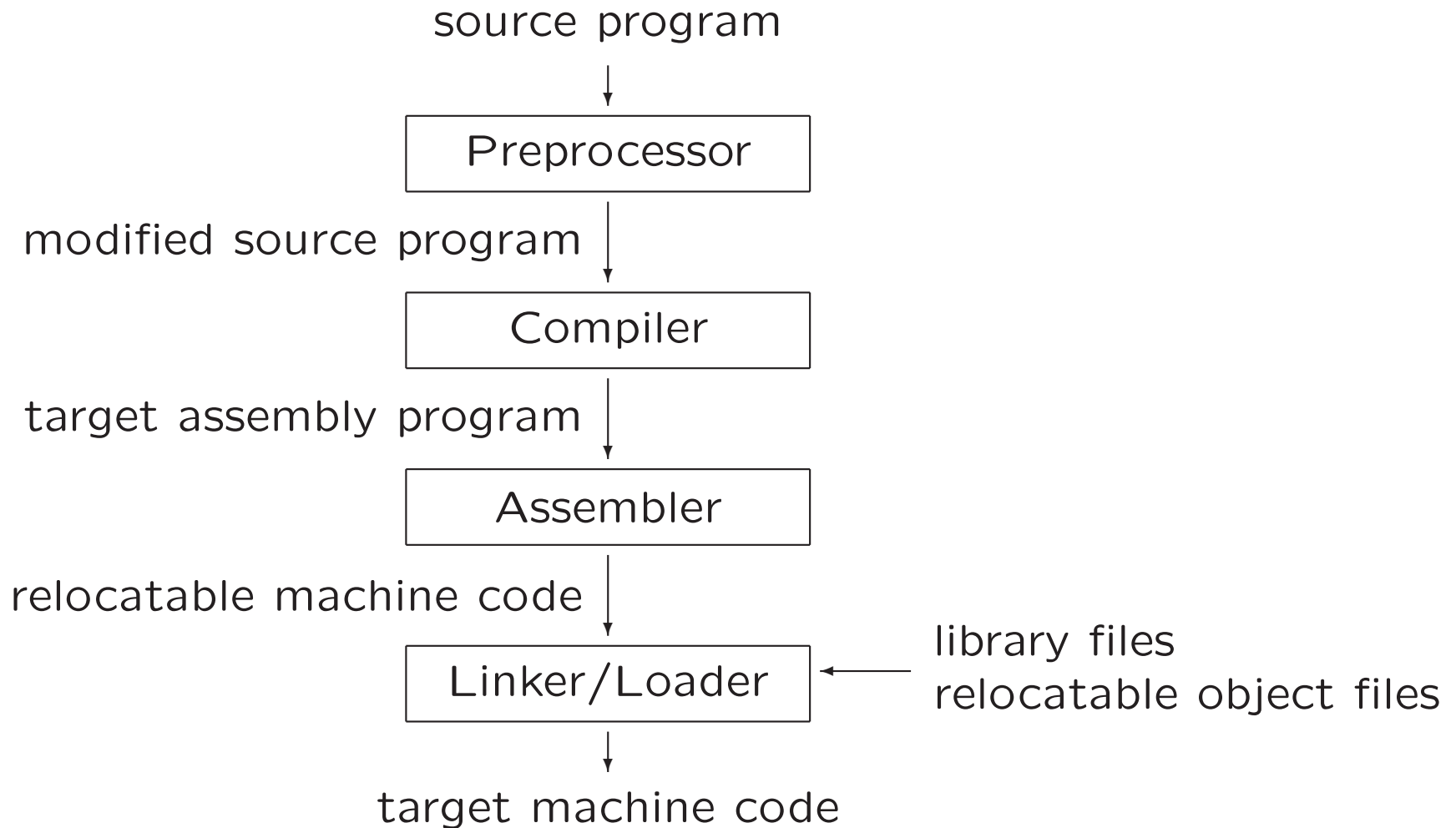
- Compiler: Translates source code into machine code, with scanner, parser, . . . , code generator
- Interpreter: Executes source code 'directly', with scanner, parser  
Statements in, e.g., a loop are scanned and parsed again and again

# 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



# Compilation flow





# 1.2 The Structure of a Compiler

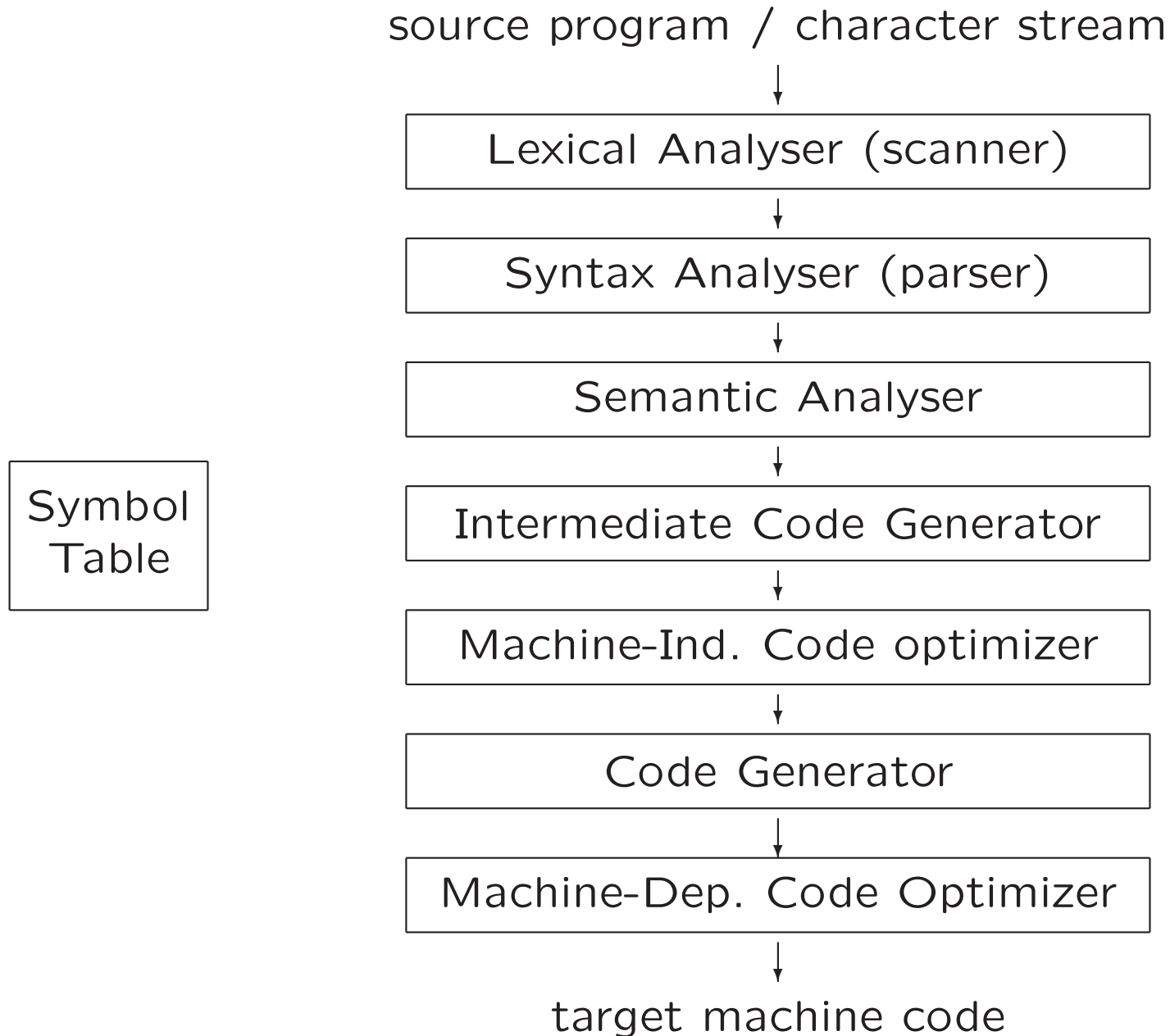
Analysis-Synthesis Model

There are two parts to compilation:

- Analysis (front end)
  - Determines the operations implied by the source program which are recorded in an intermediate representation, e.g., a tree structure
- Synthesis (back end)
  - Takes the intermediate representation and translates the operations therein into the target program

Cf. editors with syntax highlighting or text auto completion

# The Phases of a Compiler



# The Phases of a Compiler

Character stream:

```
position = initial + rate * 60
```

Lexical Analyser (scanner)

Token stream:

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

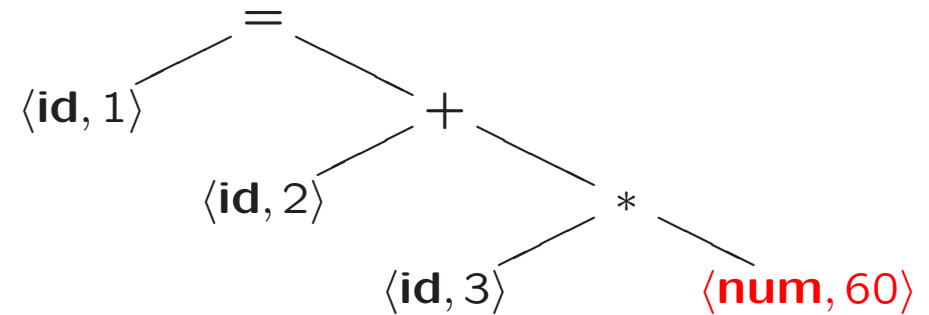
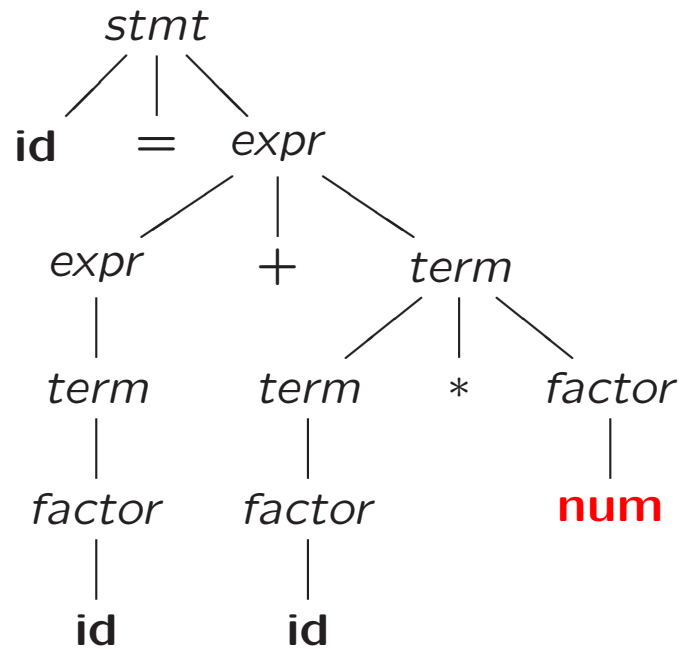
# The Phases of a Compiler

Token stream:

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

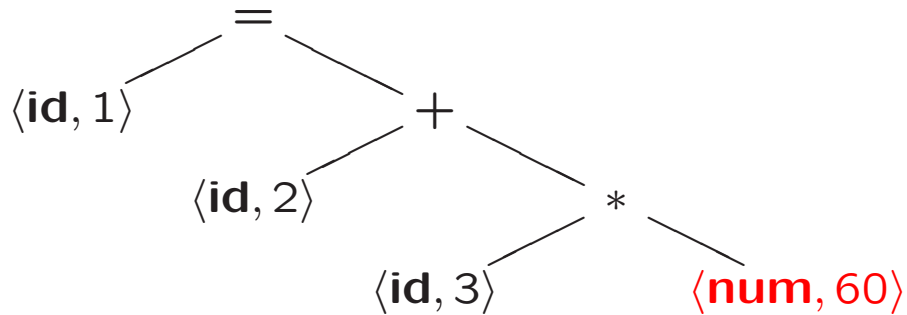
Syntax Analyser (parser)

Parse tree / syntax tree:



# The Phases of a Compiler

Syntax tree:

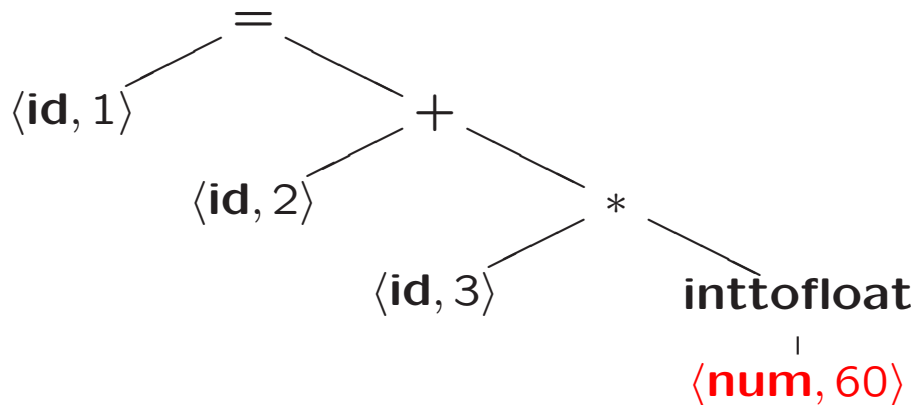


Semantic Analyser

Coercion

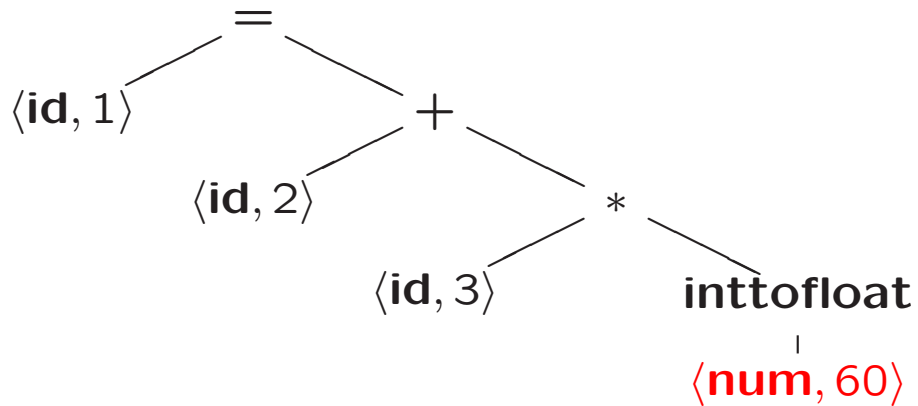
`A[i]`, `int x`, `break`, ...

Syntax tree:



# The Phases of a Compiler

Syntax tree:



Intermediate Code Generator

One operator, explicit order  
Temporary variables  
Less than three operands

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

Code Optimizer

Intermediate code (three-address code):

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

# The Phases of a Compiler

Intermediate code (three-address code):

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

Code Generator

Target code (assembly code):

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



# The Grouping of Phases

Phases constitute **logical** organization of compiler

Inefficient as implementation:

characters → Scanner → tokens → Parser → tree  
→ Semantic analyser → ... → code

**Phases are separate 'programs', which run sequentially**

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

# The Grouping of Phases

Other extreme: single-pass compiler

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

Phases work in an interleaved way

Portion of code is generated while reading portion of source program

Nowadays: often two-pass compiler

## 1.2.8 The Grouping of Phases

- Front End:  
scanning, parsing, semantic analysis, intermediate code generation  
(source code → intermediate representation)
- (optional) machine independent code optimization
- Back End:  
code generation, machine dependent code optimization  
(intermediate representation → target machine code)

**language-dependent**

Java

C

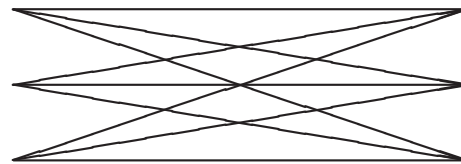
Pascal

**machine-dependent**

Pentium

PowerPC

SPARC

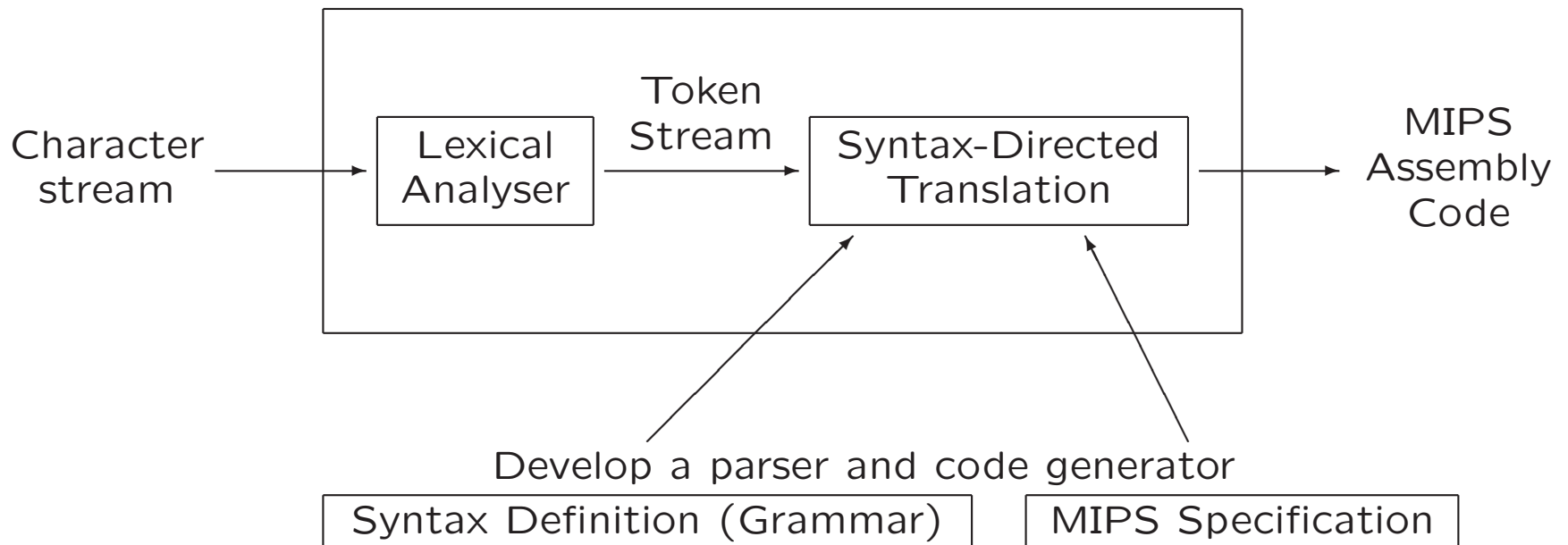


## 1.2.9 Compiler-Construction Tools

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

- Scanner generators
- Parser generators
- Syntax-directed translation engines
- Code generator generators
- Data-flow analysis engines

# The Structure of **our** compiler



Syntax-directed translation:

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

## 2.2 Syntax Definition

Context-free grammar is a 4-tuple with

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

Example: Context-free grammar for simple expressions:

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

with productions P:

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

# 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)

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

Example: 9-5+2

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

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



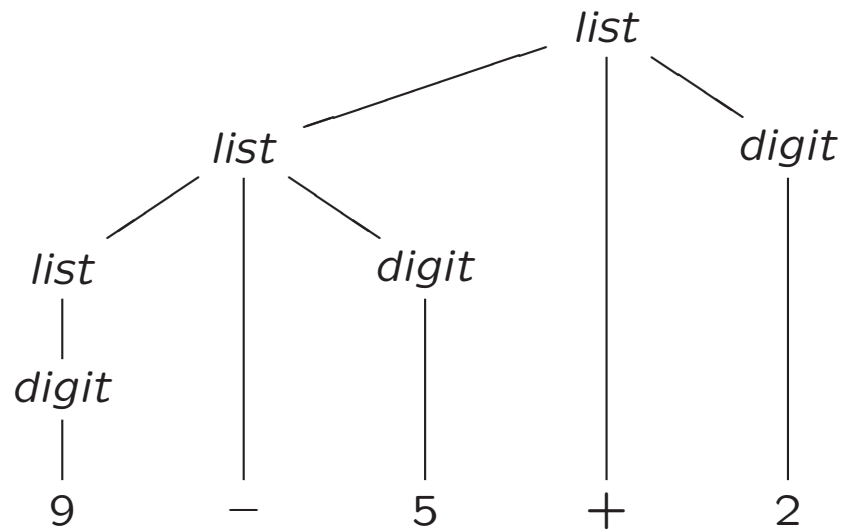
# Parse Tree

(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  $\epsilon$  (=empty)
- Each interior node is labelled by a nonterminal
- If node  $A$  has children  $X_1, X_2, \dots, X_n$ , then there must be a production  $A \rightarrow X_1 X_2 \dots X_n$

# Parse Tree (Example)

Parse tree of the string  $9 - 5 + 2$  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:

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

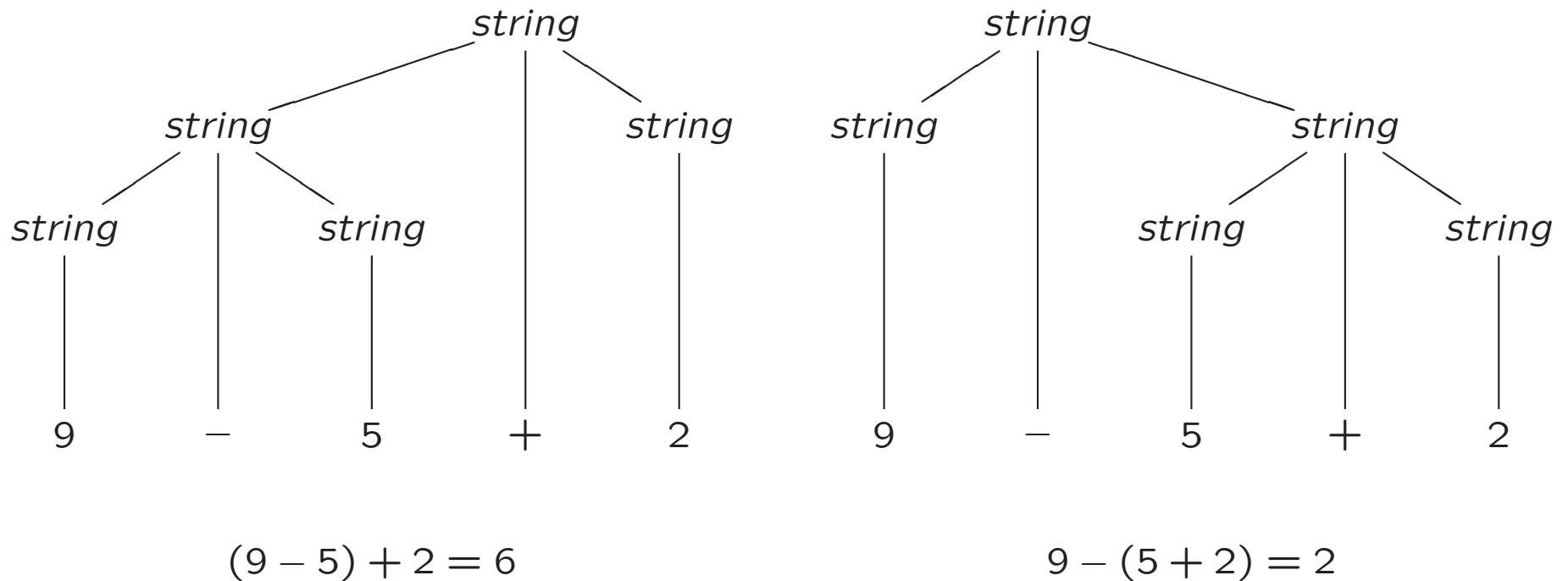
with productions  $P$

$$string \rightarrow string + string \mid string - string \mid 0 \mid 1 \mid \dots \mid 9$$

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

# Ambiguity (Example)

Parse trees of the string  $9 - 5 + 2$  using grammar  $G'$



Preferred...

# Associativity of Operators

By convention

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

In most programming languages:

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

$**$ ,  $=$  are right associative:

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

# Precedence of Operators

Consider:  $9 + 5 * 2$

Is this  $(9 + 5) * 2$  or  $9 + (5 * 2)$  ?

Associativity does not resolve this

Precedence of operators:  $\begin{array}{l} + - \\ * / \end{array} \downarrow$  increasing  
precedence

Unambiguous grammar for arithmetic expressions: ...

Example:

$$9 + 5 * 2 * 3 + 1 + 4 * 7$$

# Precedence of Operators

Consider:  $9 + 5 * 2$

Is this  $(9 + 5) * 2$  or  $9 + (5 * 2)$  ?

Associativity does not resolve this

Precedence of operators:  $\begin{array}{cc} + & - \\ * & / \end{array} \downarrow$  increasing precedence

Unambiguous grammar for arithmetic expressions:

$$\begin{aligned} \textit{expr} &\rightarrow \textit{expr} + \textit{term} \mid \textit{expr} - \textit{term} \mid \textit{term} \\ \textit{term} &\rightarrow \textit{term} * \textit{factor} \mid \textit{term} / \textit{factor} \mid \textit{factor} \\ \textit{factor} &\rightarrow \textit{digit} \mid (\textit{expr}) \\ \textit{digit} &\rightarrow 0 \mid 1 \mid \dots \mid 9 \end{aligned}$$

Parse tree for  $9 + 5 * 2 \dots$

## 2.3 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

$$\begin{aligned} \textit{expr} &\rightarrow \textit{expr}_1 + \textit{term} \mid \textit{expr}_1 - \textit{term} \mid \textit{term} \\ \textit{term} &\rightarrow 0 \mid 1 \mid \dots \mid 9 \end{aligned}$$

# Syntax-Directed Definition (Example)

Production	Semantic rule
$expr \rightarrow expr_1 + term$	$expr.t = expr_1.t \parallel term.t \parallel '+'$
$expr \rightarrow expr_1 - term$	$expr.t = expr_1.t \parallel term.t \parallel '-'$
$expr \rightarrow term$	$expr.t = term.t$
$term \rightarrow 0$	$term.t = '0'$
$term \rightarrow 1$	$term.t = '1'$
...	...
$term \rightarrow 9$	$term.t = '9'$

Result: annotated parse tree

Example:  $9 - 5 + 2$

# 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

In example, attributes contain the translated form of the input after the computations are completed  
(postfix notation corresponding to subtree)

# 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 (mainly) consider synthesized attributes

## 2.3.4 Tree Traversals

- 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

## 2.3.5 Translation Scheme

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

Example

$$\textit{expr} \rightarrow \textit{expr}_1 + \textit{term} \mid \textit{expr}_1 - \textit{term} \mid \textit{term}$$
$$\textit{term} \rightarrow 0 \mid 1 \mid \dots \mid 9$$

# Translation Scheme (Example)

$$\begin{aligned} \text{expr} &\rightarrow \text{expr}_1 + \text{term} \{\text{print('+'})\} \\ \text{expr} &\rightarrow \text{expr}_1 - \text{term} \{\text{print('-')} \} \\ \text{expr} &\rightarrow \text{term} \\ \text{term} &\rightarrow 0 \{\text{print('0')} \} \\ \text{term} &\rightarrow 1 \{\text{print('1')} \} \\ &\dots \quad \dots \\ \text{term} &\rightarrow 9 \{\text{print('9')} \} \end{aligned}$$

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

Implementation requires postorder traversal (**LRW**)



# Translations Scheme

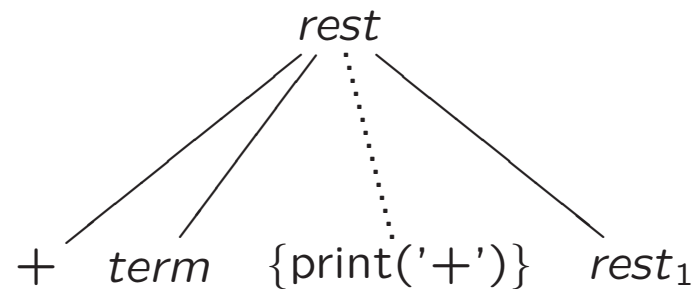
Different grammar for same expressions:

$$rest \rightarrow +term rest_1$$

With semantic action:

$$rest \rightarrow +term \{print('+')\} rest_1$$

Corresponding effect on parse tree:



## 2.4 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

# Compilerconstructie

college 1  
Overview

Chapters for reading: 1.1, 1.2, 2.1-2.3, 2.5