Compiler Construction

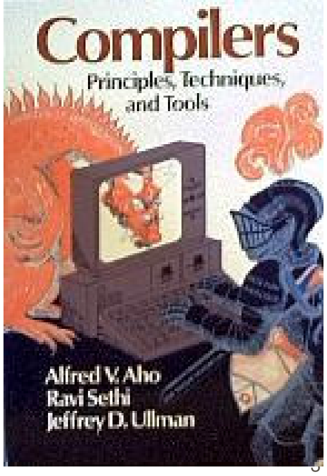
Dr. Ir. Bart Kienhuis
Computer Systems Group
LIACS

Why This Course

- Know how to build a compiler for a (simplified) (programming) language
- Know how to use compiler construction tools, such as generators for scanners and parsers
- Be able to write LL(1), LR(1) grammars (for new languages)
- Be familiar with compiler analysis and optimization techniques
- … learn how to work on a larger software project!

Course Outline

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

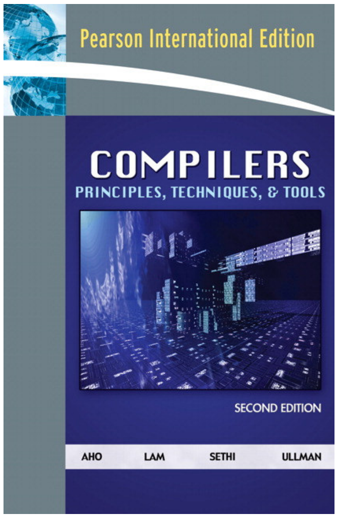


A.V. Aho, R. Sethi, en J.D. Ullman, Compilers: Principles, Techniques, and Tools, Addison-Wesley, 1986, ISBN: 0-201-10088-6.

New edition

- Dragon book has been revised in 2006
- In Second edition good improvements are made

- **Publisher:** Addison Wesley; 2 edition (August 31, 2006)
- **Language:** English
- **ISBN-10:** 0321486811



Course Outline

- Contact hours
  - Official communication medium is email
  - Blackboard (http://blackboard.leidenuniv.nl)
  - All material needed is available here
- Practicum
  - Different from previous years, we now offer 5 self contained assignments
  - These assignments are done by groups of two persons
  - Assignments are handed in into CVS

Course Outline

- Grading
  - 2 ECTS Written Exam
  - 5 ECTS Practicum
- You need to pass all 5 assignments
- No ‘late’ submissions will be accepted!
  - If you miss these assignments, you have to wait until next year

Course Outline (Tentative)

- 04/09/08 Introduction
- 11/09/08 Lexical and Syntax Analysis
- 18/09/08 Syntax Analysis assignment 1
- 25/09/08 NO CLASS
- 02/10/08 Type Checking assignment 2
- 09/10/08 Intermediate Code Generation 1
- 16/10/08 NO CLASS
- 23/10/08 Intermediate Code Generation 2 assignment 3
- 30/10/08 Code Generation 1
- 06/11/08 Code Generation 2 assignment 4
- 13/11/08 Run-Time Organization
- 20/11/08 Code Optimizations assignment 5
- 27/11/08 ELECTIVE
- 04/12/08 backup date

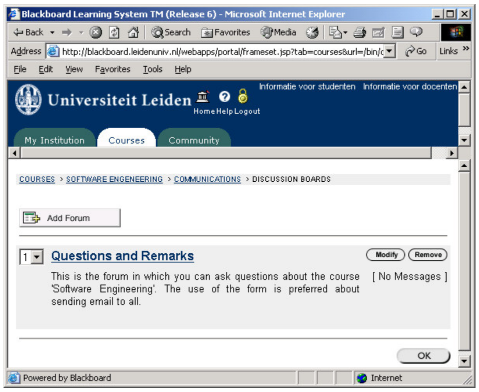
Practicum

- 28/09-02/10 assignment 1, Calculator
- 02/10-23/10 assignment 2, Parsing & Syntax tree
- 23/10-06/11 assignment 3, Intermediate code
- 06/11-20/11 assignment 4, Assembly generation
- 20/11-04/12 assignment 5, Optimizations

- All deadlines are at 17.00h (5 pm).
- The deadlines are strict.
- Submission takes place in CVS

Blackboard Coco

- Please enroll on-line to Coco in Blackboard
- Communication about Coco is shared between everyone.
- Use the 'Forum' option to ask me questions.
- If you ask me directly, I will submit also to the forum.

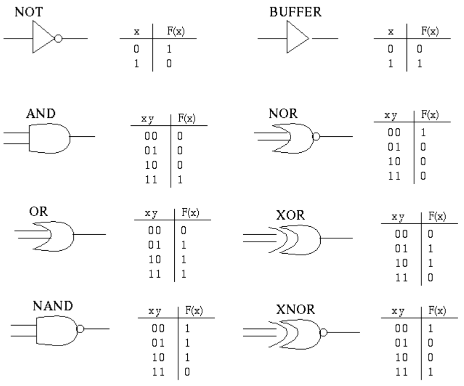


This is the forum in which you can ask questions about the course 'Software Engineering'. The use of the form is preferred about sending email to all.

Introduction

- Compiler Construction
- Missing Link between
  - Digital Technique
    - Boolean Logic
    - Flip-Flops
  - Computer Architectures
    - Memory
    - Instructions

Digital Technique



NOT

x	F(x)
0	1
1	0

BUFFER

x	F(x)
0	0
1	1

AND

x y	F(x)
0 0	0
0 1	0
1 0	0
1 1	1

NOR

x y	F(x)
0 0	1
0 1	0
1 0	0
1 1	0

OR

x y	F(x)
0 0	0
0 1	1
1 0	1
1 1	1

XOR

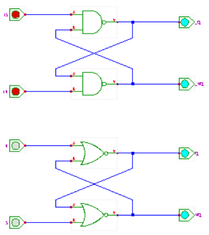
x y	F(x)
0 0	0
0 1	1
1 0	1
1 1	0

NAND

x y	F(x)
0 0	1
0 1	1
1 0	1
1 1	0

XNOR

x y	F(x)
0 0	1
0 1	0
1 0	0
1 1	1



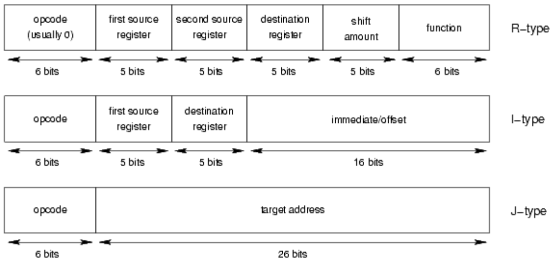
/S	/R	Q	NQ	NAND
0	0	1	1	(forbidden)
0	1	1	0	
1	0	0	1	
1	1	Q*	NQ*	(store)

S	R	Q	NQ	NOR
0	0	Q*	NQ*	(store)
0	1	0	1	
1	0	1	0	
1	1	0	0	(forbidden)

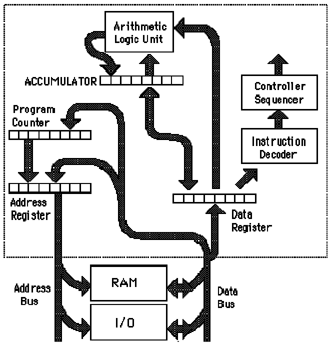
Boolean Logic

Flip Flops

Computer Architecture



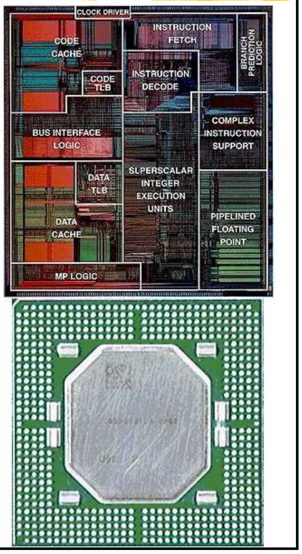
MIPS Instruction Set



Computer Architecture

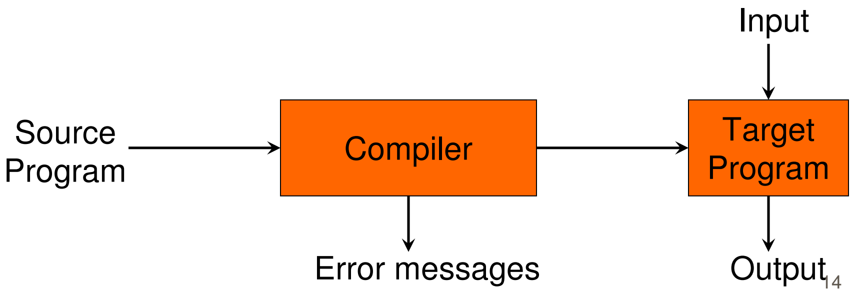
MIPS Instruction Set

opcode field	opcode	instruction format
000010	**j**	J-type
000011	**jal**	J-type
000100	**beq**	I-type
000101	**bne**	I-type
001000	**Addi**	I-type
001001	**Addiu**	I-type
001010	**Slti**	I-type
001011	**Sltiu**	I-type
001100	**Andi**	I-type



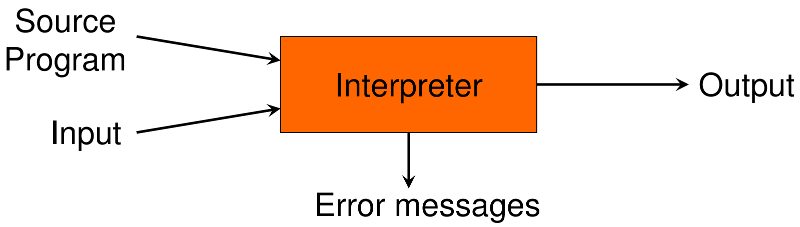
Compilers and Interpreters

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



Compilers and Interpreters (cont'd)

- "*Interpretation*"
  - Performing the operations implied by the source program



Source Program

Input

Interpreter

Output

Error messages

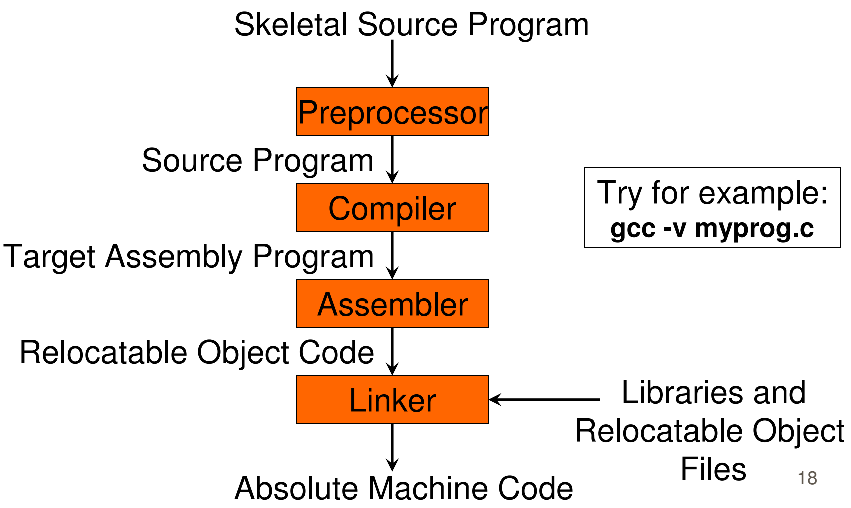
The Analysis-Synthesis Model of Compilation

- There are two parts to compilation:
  - *Analysis* determines the operations implied by the source program which are recorded in a tree structure
  - *Synthesis* takes the tree structure and translates the operations therein into the target program

Other Tools that Use the Analysis-Synthesis Model

- *Editors* (syntax highlighting)
- *Pretty printers* (e.g. doxygen)
- *Static checkers* (e.g. lint and splint)
- *Interpreters*
- *Text formatters* (e.g. TeX and LaTeX)
- *Silicon compilers* (e.g. VHDL)
- *Query interpreters/compilers* (Databases)

Preprocessors, Compilers, Assemblers, and Linkers



Skeletal Source Program

Preprocessor

Source Program

Compiler

Target Assembly Program

Assembler

Relocatable Object Code

Linker

Libraries and Relocatable Object Files

Absolute Machine Code

Try for example:
gcc -v myprog.c

The Phases of a Compiler

Phase	Output	Sample
Programmer	Source string	`A=B+C;`
Scanner (performs *lexical analysis*)	Token string	`'A', '=', 'B', '+', 'C', ';'` And *symbol table* for identifiers

Parser (performs *syntax analysis* based on the grammar of the programming language)	Parse tree or abstract syntax tree	`|`
		`=`
		`/ \`
		`A   +`
		`/ \`
		`B   C`

Semantic analyzer (type checking, etc)	Parse tree or abstract syntax tree	

Intermediate code generator	Three-address code, quads, or RTL	`int2fp B      t1`
		`+      t1  C  t2`
		`:=     t2     A`

Optimizer	Three-address code, quads, or RTL	`int2fp B      t1` `+      t1 #2.3 A`

Code generator	Assembly code	`MOVF  #2.3,r1`
		`ADDF2 r1,r2`
		`MOVF  r2,A`

Peephole optimizer	Assembly code	`ADDF2 #2.3,r2` `MOVF  r2,A`

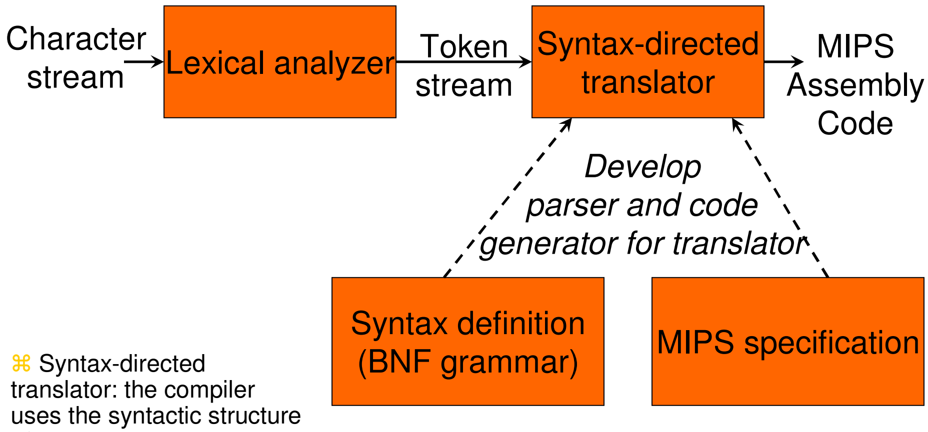
The Grouping of Phases

- Compiler front and back ends:
  - Analysis (*machine independent* front end)
  - Synthesis (*machine dependent* back end)
- Passes
  - A collection of phases may be repeated only once (*single pass*) or multiple times (*multi pass*)
  - Single pass: usually requires everything to be defined before being used in source program
  - Multi pass: compiler may have to keep entire program representation in memory

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 translator: the compiler uses the syntactic structure of the language to generate output

Syntax Definition

- Context-free grammar is a 4-tuple with
  - A set of tokens (*terminal* symbols)
  - A set of *nonterminals*
  - A set of *productions*
  - A designated *start symbol*

Example Grammar

Context-free grammar for simple expressions:

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

with productions P =

$$list \rightarrow list \mathbf{+} digit$$

$$list \rightarrow list \mathbf{-} digit$$

$$list \rightarrow digit$$

$$digit \rightarrow \mathbf{0} \mid \mathbf{1} \mid \mathbf{2} \mid \mathbf{3} \mid \mathbf{4} \mid \mathbf{5} \mid \mathbf{6} \mid \mathbf{7} \mid \mathbf{8} \mid \mathbf{9}$$

Derivation

- Given a CF grammar we can determine the set of all *strings* (sequences of tokens) generated by the grammar using *derivation*
  - We begin with the start symbol
  - In each step, we replace one nonterminal in the current *sentential form* with one of the right-hand sides of a production for that nonterminal

Derivation for the Example Grammar

list
⇒ *list* + *digit*
⇒ *list* **-** *digit* + *digit*
⇒ *digit* **-** *digit* + *digit*
⇒ **9 -** *digit* + *digit*
⇒ **9 - 5 +** *digit*
⇒ **9 - 5 + 2**

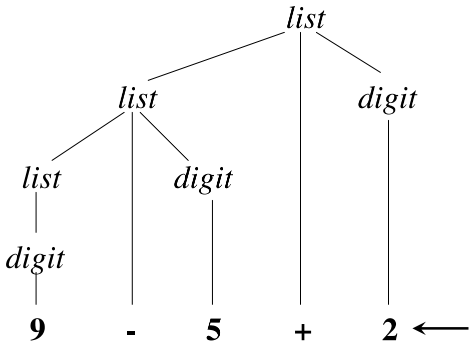
This is an example *leftmost derivation*, because we replaced the leftmost nonterminal (underlined) in each step

Parse Trees

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

Parse Tree for the Example Grammar

Parse tree of the string **9-5+2** using grammar *G*



The sequence of leafs is called the *yield* of the parse tree

Ambiguity

Consider the following context-free grammar:

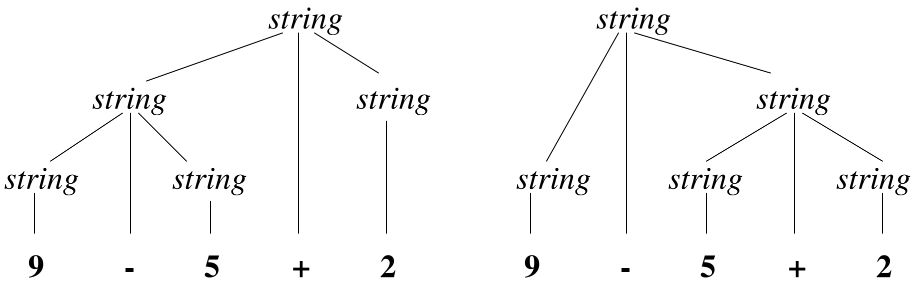
$G = <\{string\}, \{\mathbf{+,-,0,1,2,3,4,5,6,7,8,9}\}, P, string>$

with production P =

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

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

Ambiguity (cont'd)



Associativity of Operators

Left-associative operators have *left-recursive* productions

left → *left* + *term* | *term*

String **a+b+c** has the same meaning as **(a+b)+c**

Right-associative operators have *right-recursive* productions

right → *term* = *right* | *term*

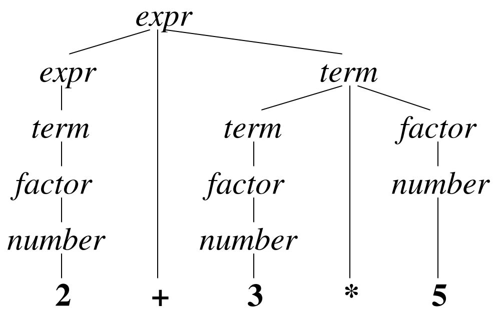
String **a=b=c** has the same meaning as **a=(b=c)**

Precedence of Operators

Operators with higher precedence “bind more tightly”

expr → *expr* + *term* | *term*
term → *term* * *factor* | *factor*
factor → *number* | (*expr*)

String **2+3*5** has the same meaning as **2+(3*5)**



Syntax of Statements

stmt → **id** **:=** *expr*
| **if** *expr* **then** *stmt*
| **if** *expr* **then** *stmt* **else** *stmt*
| **while** *expr* **do** *stmt*
| **begin** *opt_stmts* **end**
opt_stmts → *stmt* **;** *opt_stmts*
| ε

Syntax-Directed Translation

- Uses a CF grammar to specify the syntactic structure of the language
- AND associates a set of *attributes* with (non)terminals
- AND 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

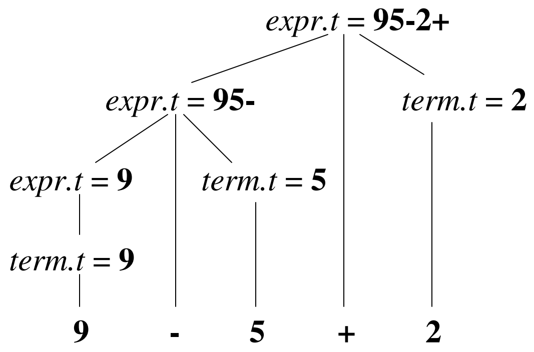
Synthesized and Inherited Attributes

- An attribute is said to be …
  - *synthesized* if its value at a parse-tree node is determined from the attribute values at the children of the node
  - *inherited* if its value at a parse-tree node is determined by the parent (by enforcing the parent’s semantic rules)

Example Attribute Grammar

Production	Semantic Rule
$expr \rightarrow expr_1 + term$	$expr.t := expr_1.t \Vert term.t \Vert$ “+”
$expr \rightarrow expr_1 - term$	$expr.t := expr_1.t \Vert term.t \Vert$ “-”
$expr \rightarrow term$	$expr.t := term.t$
$term \rightarrow \mathbf{0}$	$term.t :=$ “0”
$term \rightarrow \mathbf{1}$	$term.t :=$ “1”
…	…
$term \rightarrow \mathbf{9}$	$term.t :=$ “9”

Example Annotated Parse Tree



expr.t = **95-2+**

expr.t = **95-**

term.t = **2**

expr.t = **9**

term.t = **5**

term.t = **9**

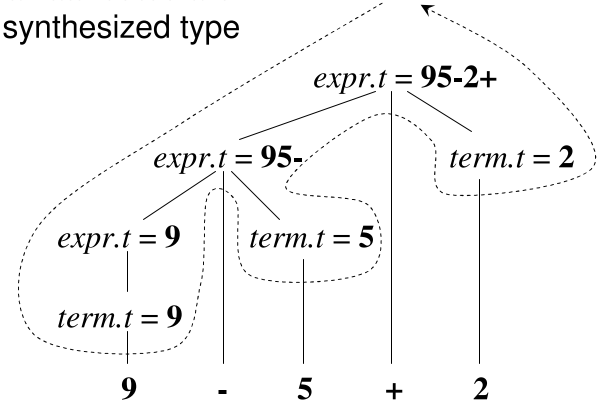
9 - 5 + 2

Depth-First Traversals

```
procedure visit(n : node);
begin
   for each child m of n, from left to right do
      visit(m);
   evaluate semantic rules at node n
end
```

Depth-First Traversals (Example)

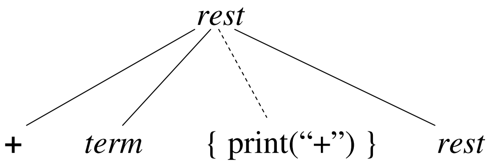
Note: all attributes are
of the synthesized type



Translation Schemes

- A *translation scheme* is a CF grammar embedded with *semantic actions*

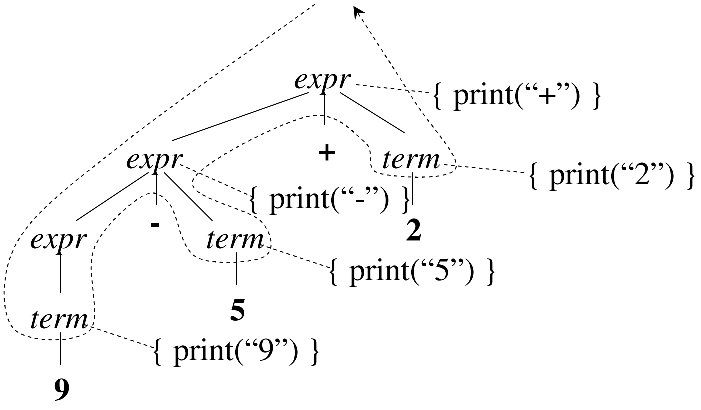
$$rest \rightarrow + \; term \; \underbrace{\{ \text{ print("+") } \}}_{\text{Embedded semantic action}} \; rest$$



Example Translation Scheme

expr → *expr* **+** *term*	{ print("+") }
expr → *expr* **-** *term*	{ print("-") }
expr → *term*	
term → **0**	{ print("0") }
term → **1**	{ print("1") }
…	…
term → **9**	{ print("9") }

Example Translation Scheme (cont'd)



Translates **9-5+2** into postfix **95-2+**

Parsing

- Parsing = *process of determining if a string of tokens can be generated by a grammar*
- For any CF grammar there is a parser that takes at most $O(n^3)$ time to parse a string of n tokens
- Linear algorithms suffice for parsing programming language
- *Top-down parsing* "constructs" parse tree from root to leaves
- *Bottom-up parsing* "constructs" parse tree from leaves to root

Predictive Parsing

- *Recursive descent parsing* is a top-down parsing method
  - Every nonterminal has one (recursive) procedure responsible for parsing the nonterminal's syntactic category of input tokens
  - When a nonterminal has multiple productions, each production is implemented in a branch of a selection statement based on input look-ahead information
- *Predictive parsing* is a special form of recursive descent parsing where we use one lookahead token to unambiguously determine the parse operations

Example Predictive Parser (Grammar)

type → *simple*
| **^ id**
| **array [** *simple* **] of** *type*
simple → **integer**
| **char**
| **num dotdot num**

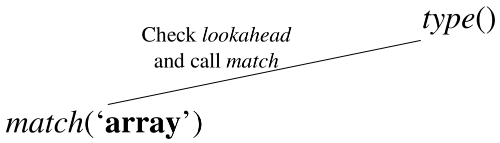
Example Predictive Parser (Program Code)

```
procedure match(t : token);
begin
   if lookahead = t then
      lookahead := nexttoken()
   else error()
end;

procedure type();
begin
   if lookahead in { ‘integer’, ‘char’, ‘num’ } then
      simple()
   else if lookahead = ‘^’ then
      match(‘^’); match(id)
   else if lookahead = ‘array’ then
      match(‘array’); match(‘[‘); simple();
      match(‘]’); match(‘of’); type()
   else error()
end;
```

```
procedure simple();
begin
   if lookahead = ‘integer’ then
      match(‘integer’)
   else if lookahead = ‘char’ then
      match(‘char’)
   else if lookahead = ‘num’ then
      match(‘num’);
      match(‘dotdot’);
      match(‘num’)
   else error()
end;
```

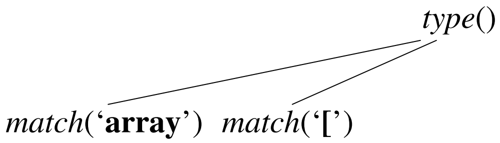
Example Predictive Parser (Execution Step 1)



Input: **array** **[** **num** **dotdot** **num** **]** **of** **integer**

↑ *lookahead*

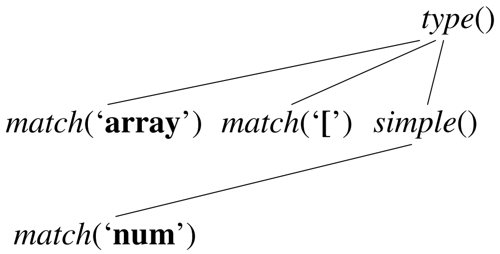
Example Predictive Parser (Execution Step 2)



Input: **array** **[** **num** **dotdot** **num** **]** **of** **integer**

↑ *lookahead*

Example Predictive Parser (Execution Step 3)



type()

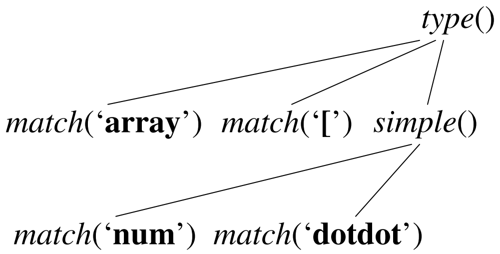
match(‘**array**’) *match*(‘**[**’) *simple*()

match(‘**num**’)

Input: **array [num dotdot num] of integer**

↑ *lookahead* (at **num**)

Example Predictive Parser (Execution Step 4)



type()

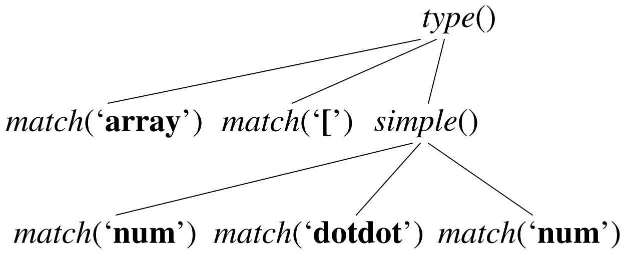
match(‘**array**’) *match*(‘**[**’) *simple*()

match(‘**num**’) *match*(‘**dotdot**’)

Input: **array [num dotdot num] of integer**

↑ *lookahead* (at **dotdot**)

Example Predictive Parser (Execution Step 5)



type()

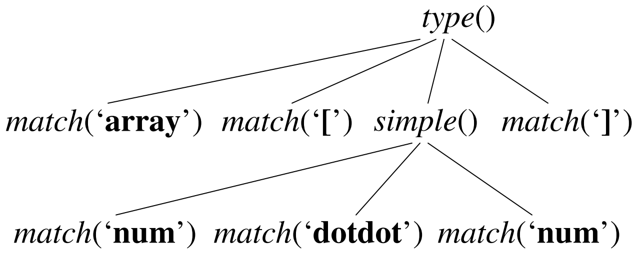
match(‘**array**’) *match*(‘[’) *simple*()

match(‘**num**’) *match*(‘**dotdot**’) *match*(‘**num**’)

Input: **array [num dotdot num] of integer**

↑ *lookahead*

Example Predictive Parser (Execution Step 6)



type()

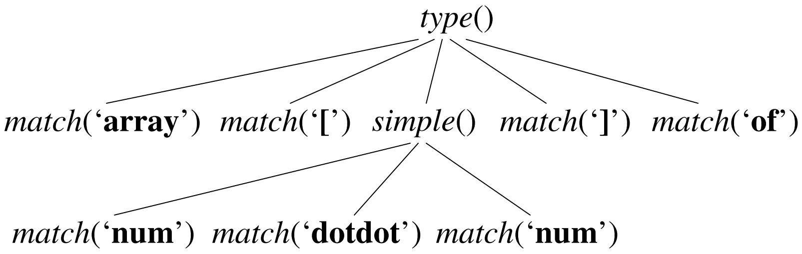
match(‘**array**’) *match*(‘[’) *simple*() *match*(‘]’)

match(‘**num**’) *match*(‘**dotdot**’) *match*(‘**num**’)

Input: **array [num dotdot num] of integer**

↑ *lookahead*

Example Predictive Parser (Execution Step 7)

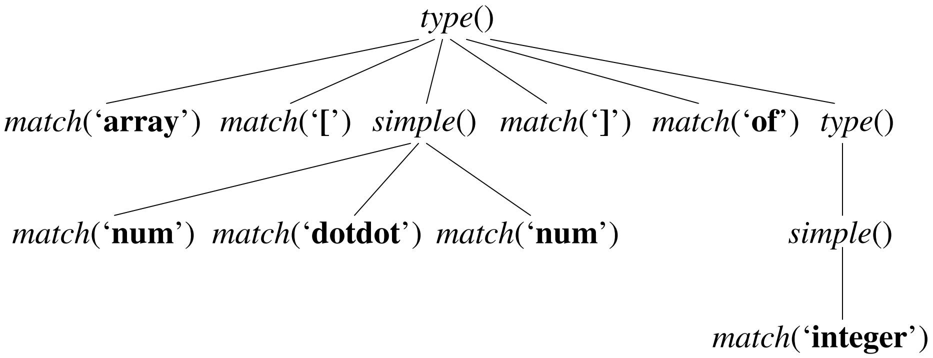


Input: **array [num dotdot num] of integer**

↑

lookahead

Example Predictive Parser (Execution Step 8)



Input: **array [num dotdot num] of integer**

↑

lookahead

FIRST

FIRST(α) is the set of terminals that appear as the first symbols of one or more strings generated from α

$$
\begin{aligned}
\textit{type} &\rightarrow \textit{simple} \\
&\mid \ \hat{}\ \textbf{id} \\
&\mid \textbf{array [} \textit{simple} \textbf{] of } \textit{type} \\
\textit{simple} &\rightarrow \textbf{integer} \\
&\mid \textbf{char} \\
&\mid \textbf{num dotdot num}
\end{aligned}
$$

FIRST(*simple*) = { **integer**, **char**, **num** }
FIRST(**^ id**) = { **^** }
FIRST(*type*) = { **integer**, **char**, **num**, **^**, **array** }

Using FIRST

We use FIRST to write a predictive parser as follows

$$
\begin{aligned}
\textit{expr} &\rightarrow \textit{term rest} \\
\textit{rest} &\rightarrow \textbf{+}\ \textit{term rest} \\
&\mid \textbf{-}\ \textit{term rest} \\
&\mid \varepsilon
\end{aligned}
$$

```
procedure rest();
begin
   if lookahead in FIRST(+ term rest) then
      match('+'); term(); rest()
   else if lookahead in FIRST(- term rest) then
      match('-'); term(); rest()
   else return
end;
```

When a nonterminal *A* has two (or more) productions as in

$$
\begin{aligned}
A &\rightarrow \alpha \\
&\mid \beta
\end{aligned}
$$

Then FIRST (α) and FIRST(β) must be disjoint for predictive parsing to work

Left Factoring

When more than one production for nonterminal *A* starts with the same symbols, the FIRST sets are not disjoint

stmt → **if** *expr* **then** *stmt*
| **if** *expr* **then** *stmt* **else** *stmt*

We can use *left factoring* to fix the problem

stmt → **if** expr **then** *stmt opt_else*
opt_else → **else** *stmt*
| ε

Left factoring: if not clear what to chose, rewrite the production until we have seen enough to make a decision.

Left Recursion

When a production for nonterminal *A* starts with a *self reference* then a predictive parser loops forever

$$\begin{aligned} A &\rightarrow A\,\alpha \\ &\mid \beta \\ &\mid \gamma \end{aligned}$$

We can eliminate *left recursive productions* by systematically rewriting the grammar using *right recursive productions*

$$\begin{aligned} A &\rightarrow \beta\, R \\ &\mid \gamma\, R \\ R &\rightarrow \alpha\, R \\ &\mid \varepsilon \end{aligned}$$

A Translator for Simple Expressions

expr → *expr* **+** *term* { print(“+”) }
expr → *expr* **-** *term* { print(“-”) }
expr → *term*
term → **0** { print(“0”) }
term → **1** { print(“1”) }
… …
term → **9** { print(“9”) }

After left recursion elimination:

expr → *term rest*
rest → **+** *term* { print(“+”) } *rest* | **-** *term* { print(“-”) } *rest* | ε
term → **0** { print(“0”) }
term → **1** { print(“1”) }
…
term → **9** { print(“9”) }

```
main()
{   lookahead = getchar();
    expr();
}
```

expr → *term rest*

rest → **+** *term* { print(“+”) } *rest*
| **-** *term* { print(“-”) } *rest*
| ε

```
expr()
{   term();
    while (1) /* optimized by inlining rest()
                 and removing recursive calls */
    {   if (lookahead == ‘+’)
        {   match(‘+’); term(); putchar(‘+’);
        }
        else if (lookahead == ‘-’)
        {   match(‘-’); term(); putchar(‘-’);
        }
        else break;
    }
}
```

term → **0** { print(“0”) }
term → **1** { print(“1”) }
…
term → **9** { print(“9”) }

```
term()
{   if (isdigit(lookahead))
    {   putchar(lookahead); match(lookahead);
    }
    else error();
}
match(int t)
{   if (lookahead == t)
        lookahead = getchar();
    else error();
}
error()
{   printf(“Syntax error\n”);
    exit(1);
}
```