Principles of Programming Languages

http://www.di.unipi.it/~andrea/Didattica/PLP-14/

Prof. Andrea Corradini
Department of Computer Science, Pisa

Lesson 2

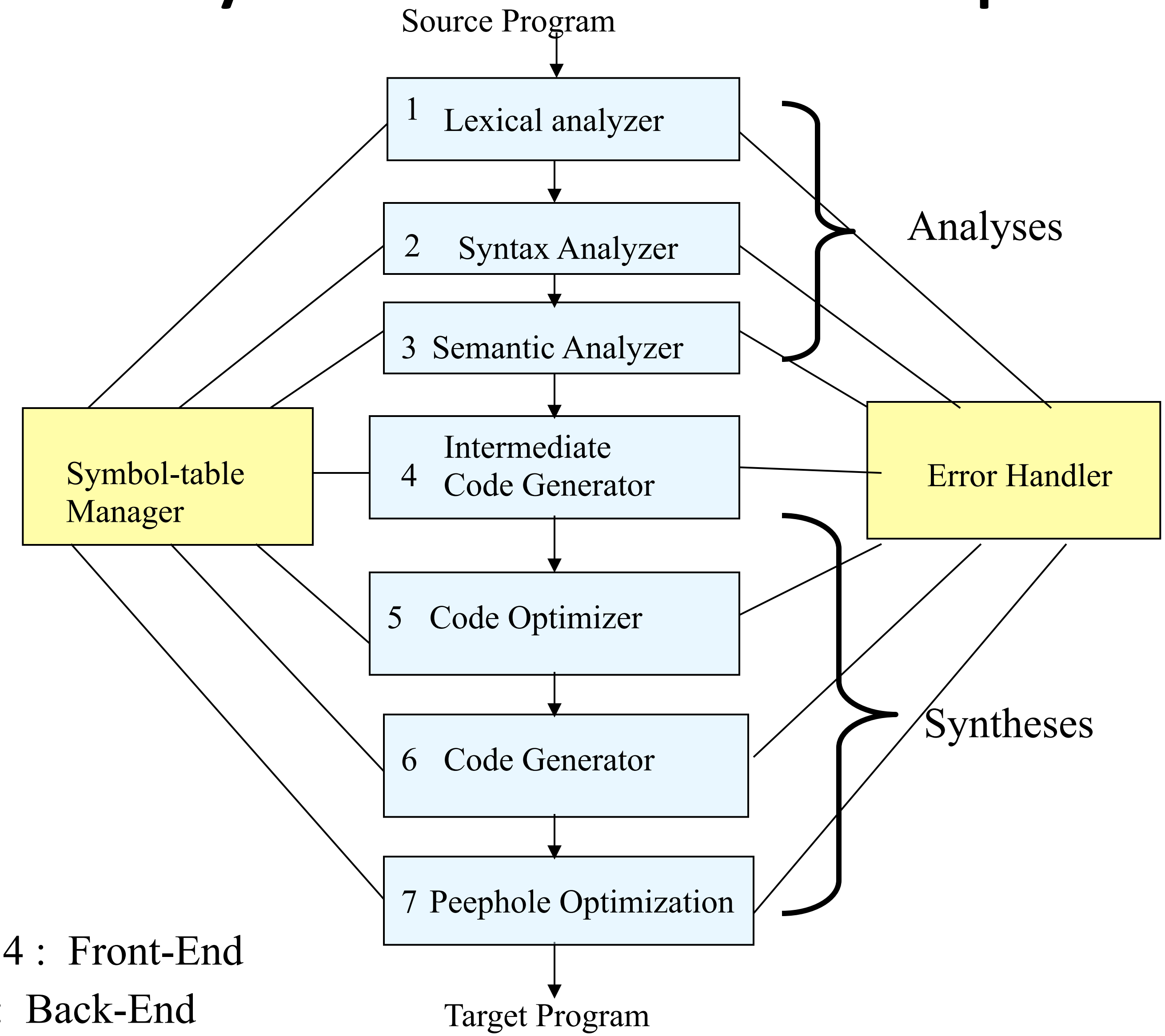
- The structure of a compiler
- Overview of a Simple Compiler Front-end
  - Predictive top-down parsing
  - Syntax directed translation
  - Lexical analysis

Admins

- Office Hours:
  - Wednesday, 9 - 11 ← my proposal
  - *Monday, 18 - 19:30*
  - *Friday, 9-11*

- Check your data and add the University ID (matricola) in the sheet

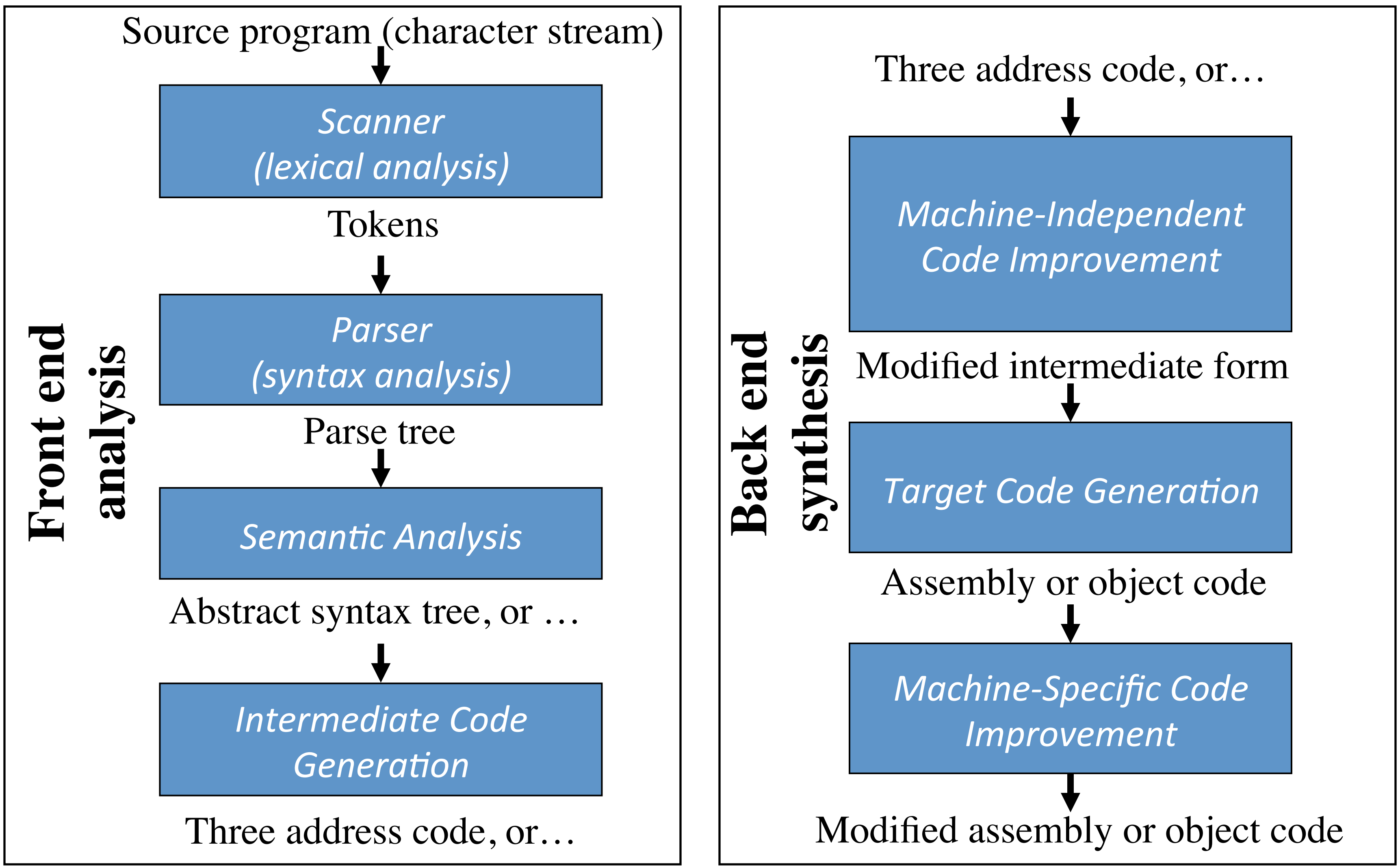
The Many Phases of a Compiler



1, 2, 3, 4 : Front-End

5, 6, 7 : Back-End

Compiler Front- and Back-end



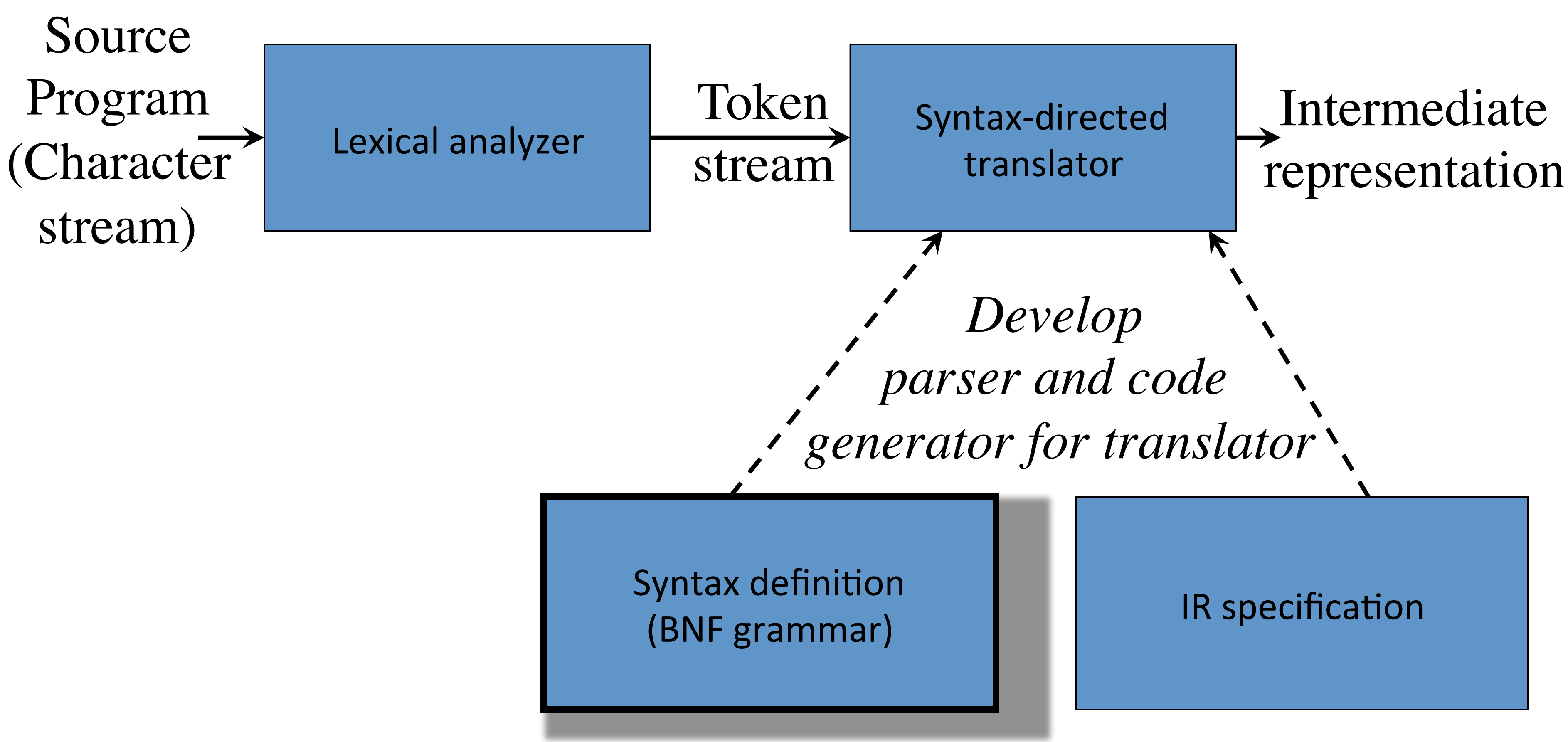
Single-pass vs. Multi-pass Compilers

- A collection of compilation phases is done only once (*single pass*) or multiple times (*multi pass*)
- **Single pass**: more efficient and uses less memory
  - requires everything to be defined before being used
  - standard for languages like Pascal, FORTRAN, C
  - Influenced the design of early programming languages
- **Multi pass**: needs more memory (to keep entire program), usually slower
  - needed for languages where declarations e.g. of variables may follow their use (Java, ADA, …)
  - allows better optimization of target code

Overview of a Simple Compiler Front-end

- Building a compiler involves:
  - Defining the *syntax* of a programming language
  - Develop a source code parser: we consider here *predictive parsing*
  - Implementing *syntax directed translation* to generate intermediate code

The Structure of the Front-End



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:

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

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**

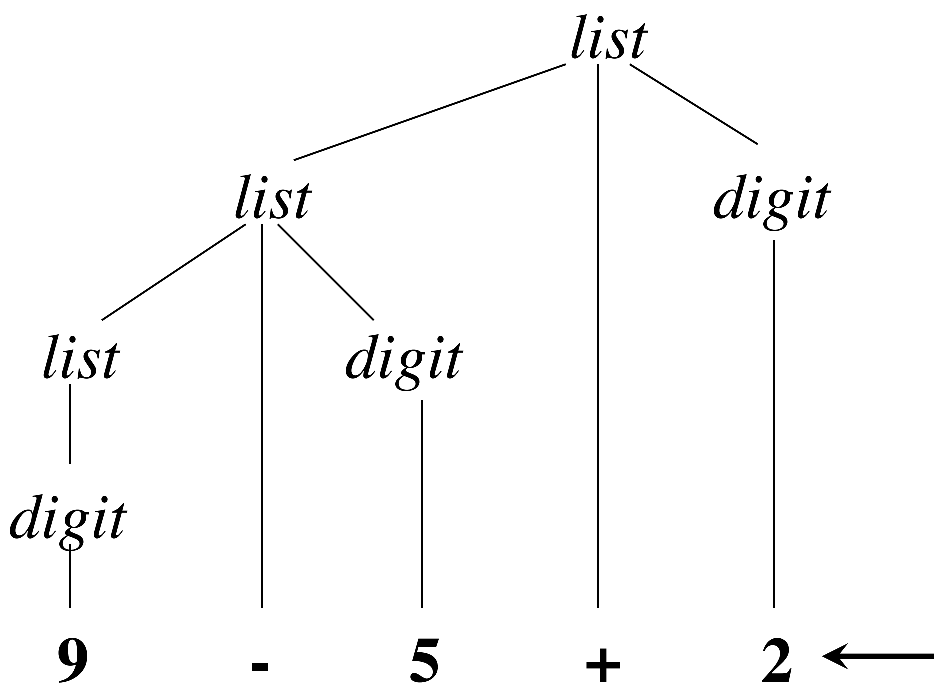
This is an example *leftmost derivation*, because we replaced the leftmost nonterminal (underlined) in each step. Likewise, a *rightmost derivation* replaces the rightmost nonterminal in each step

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
- 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 immediate *children* $X_1, X_2, \ldots, X_n$ where X_i is a (non)terminal or ε (ε denotes the *empty string*)

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:

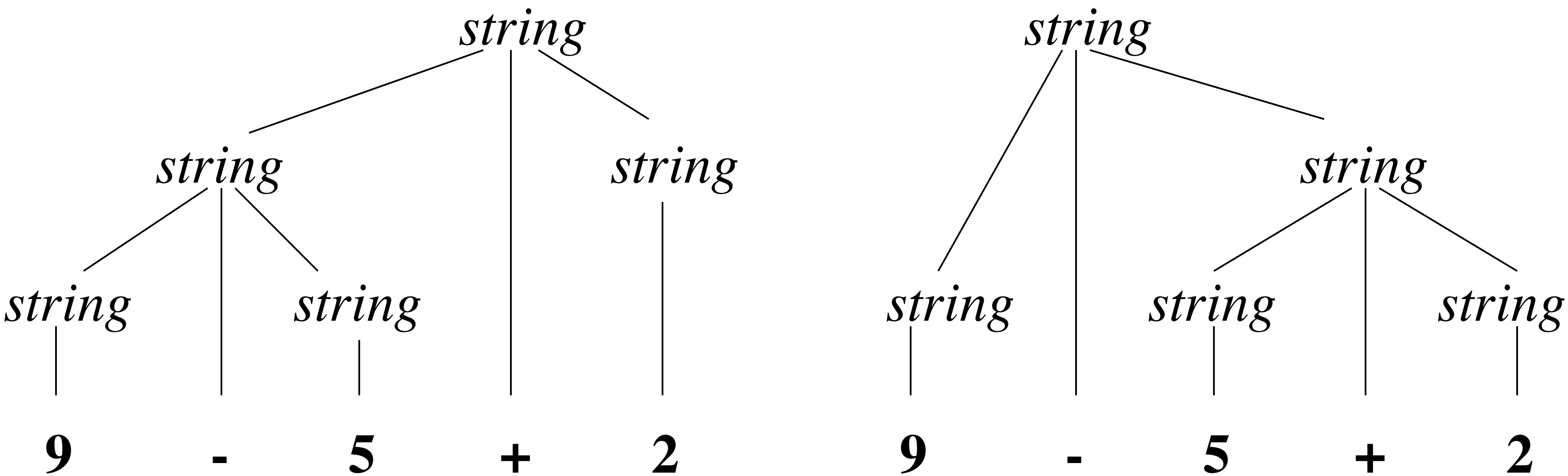
$$G = \langle\{string\}, \{\mathbf{+},\mathbf{-},\mathbf{0},\mathbf{1},\mathbf{2},\mathbf{3},\mathbf{4},\mathbf{5},\mathbf{6},\mathbf{7},\mathbf{8},\mathbf{9}\}, P, string\rangle$$

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 represents the string **9-5+2**

Ambiguity (cont'd)



Associativity of Operators

Left-associative operators have *left-recursive* productions

$$left \rightarrow left \ \mathbf{+} \ term \mid term$$

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

Right-associative operators have *right-recursive* productions

$$right \rightarrow term \ \mathbf{=} \ right \mid term$$

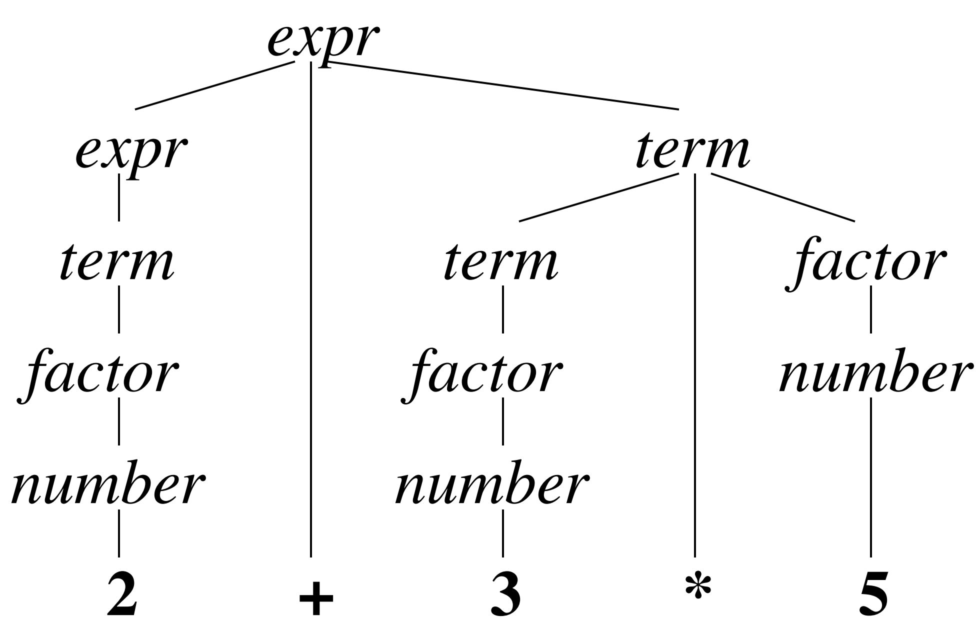
String **a=b=c** has the same meaning as **a=(b=c)**

Precedence of Operators

Operators with higher precedence "bind more tightly"

$$expr \rightarrow expr \;\mathbf{+}\; term \mid term$$
$$term \rightarrow term \;\mathbf{*}\; factor \mid factor$$
$$factor \rightarrow number \mid \mathbf{(}\; expr \;\mathbf{)}$$

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

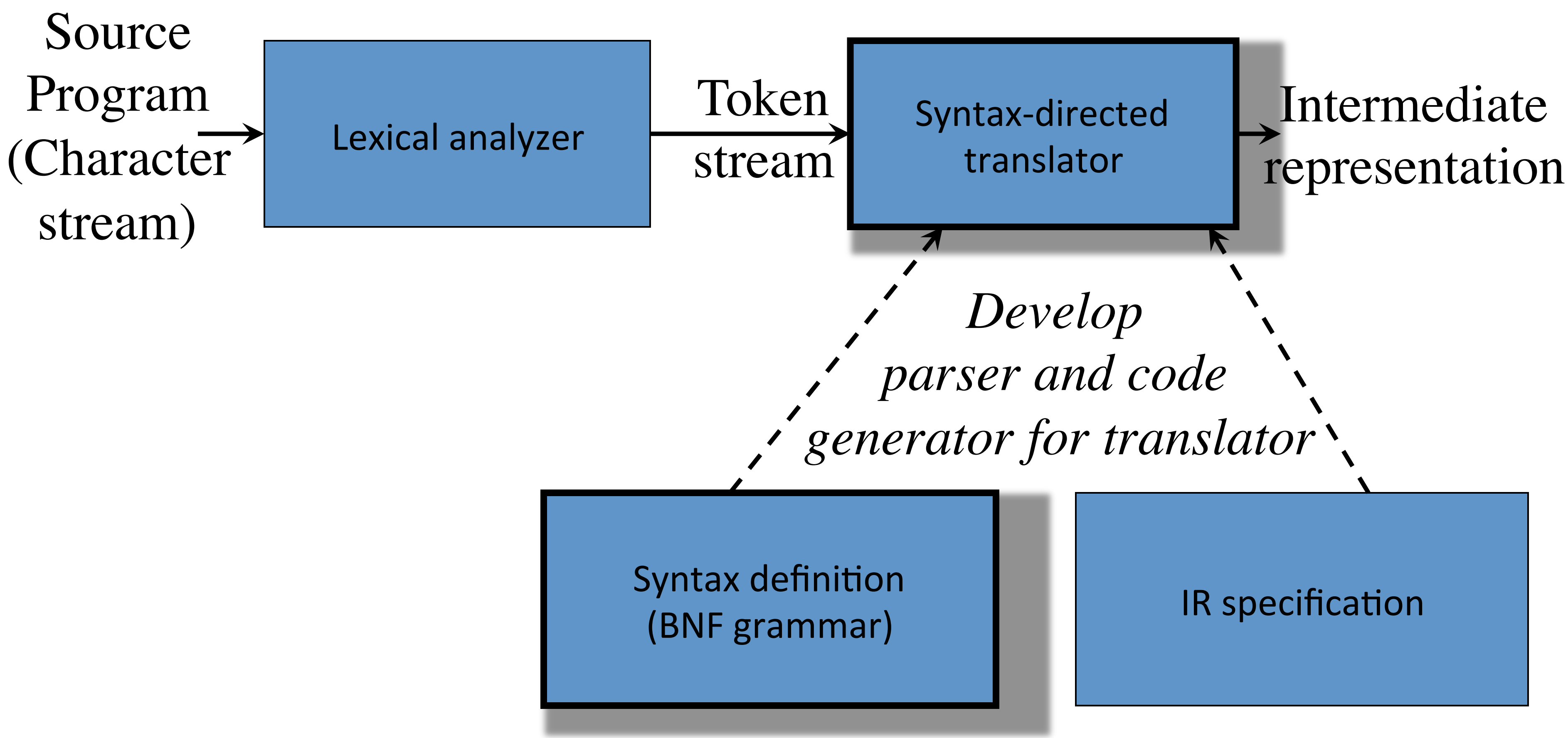


Syntax of Statements

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

The Structure of the Front-End



Syntax-Directed Translation

- Uses a CF grammar to specify the syntactic structure of the language
- AND associates a set of *attributes* with the terminals and nonterminals of the grammar
- AND associates with each production a set of *semantic rules* to compute values of attributes
- A parse tree is traversed and semantic rules applied: after the tree traversal(s) are completed, the attribute values on the nonterminals contain the translated form of the input

Synthesized and Inherited Attributes

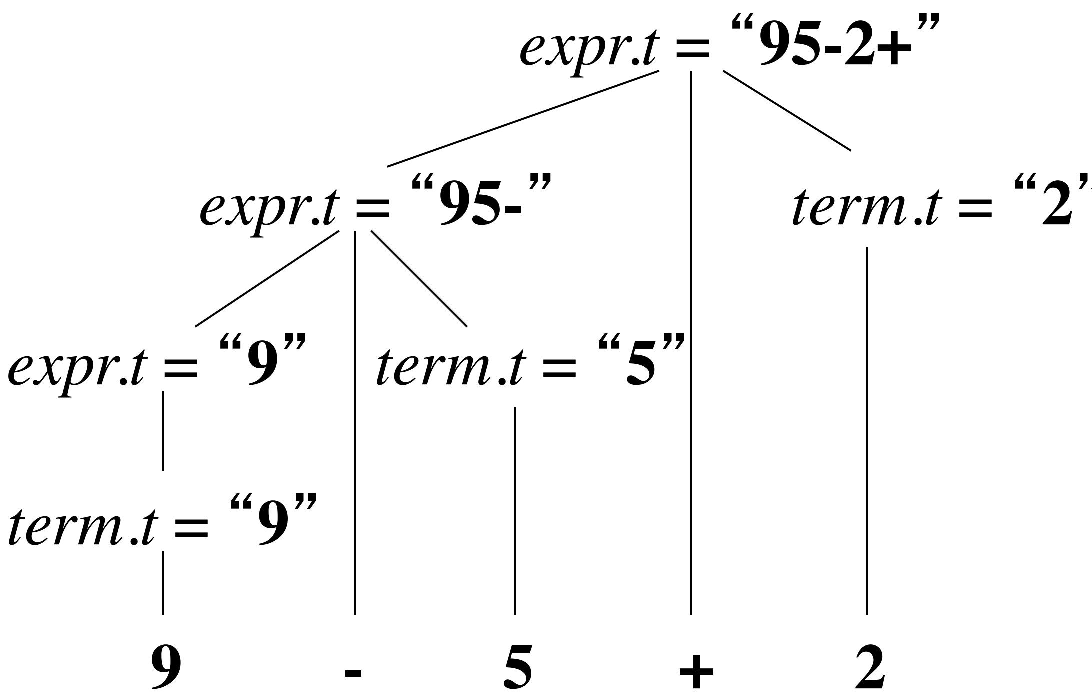
- An attribute is said to be …
  - *synthesized* if its value at a parse-tree node is determined from the attribute values at the children of the node
  - *inherited* if its value at a parse-tree node is determined by the parent (by enforcing the parent's semantic rules)

Example Attribute Grammar (Postfix Form)

String concat operator

Production	Semantic Rule
$expr \rightarrow expr_1$ **+** $term$	$expr.t := expr_1.t$ // $term.t$ // "+"
$expr \rightarrow expr_1$ **-** $term$	$expr.t := expr_1.t$ // $term.t$ // "-"
$expr \rightarrow term$	$expr.t := term.t$
$term \rightarrow$ **0**	$term.t :=$ "0"
$term \rightarrow$ **1**	$term.t :=$ "1"
…	…
$term \rightarrow$ **9**	$term.t :=$ "9"

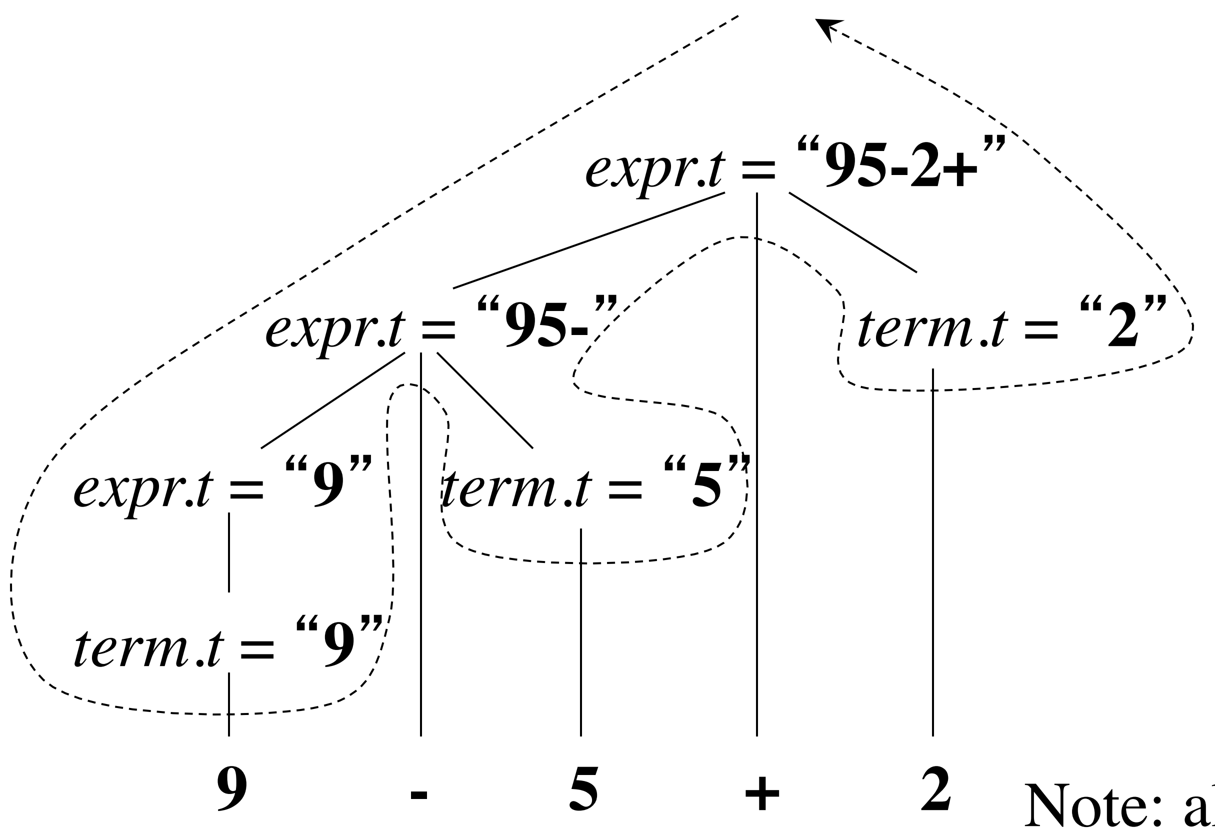
Example Annotated Parse Tree



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)



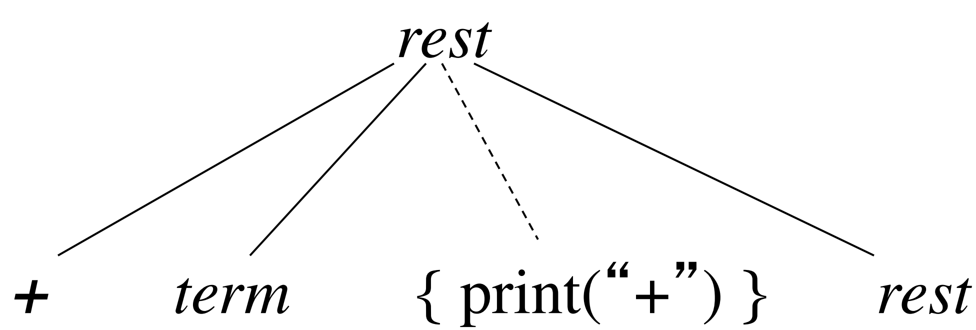
Note: all attributes are of the synthesized type

Translation Schemes

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

rest → **+** *term* { print("+") } *rest*

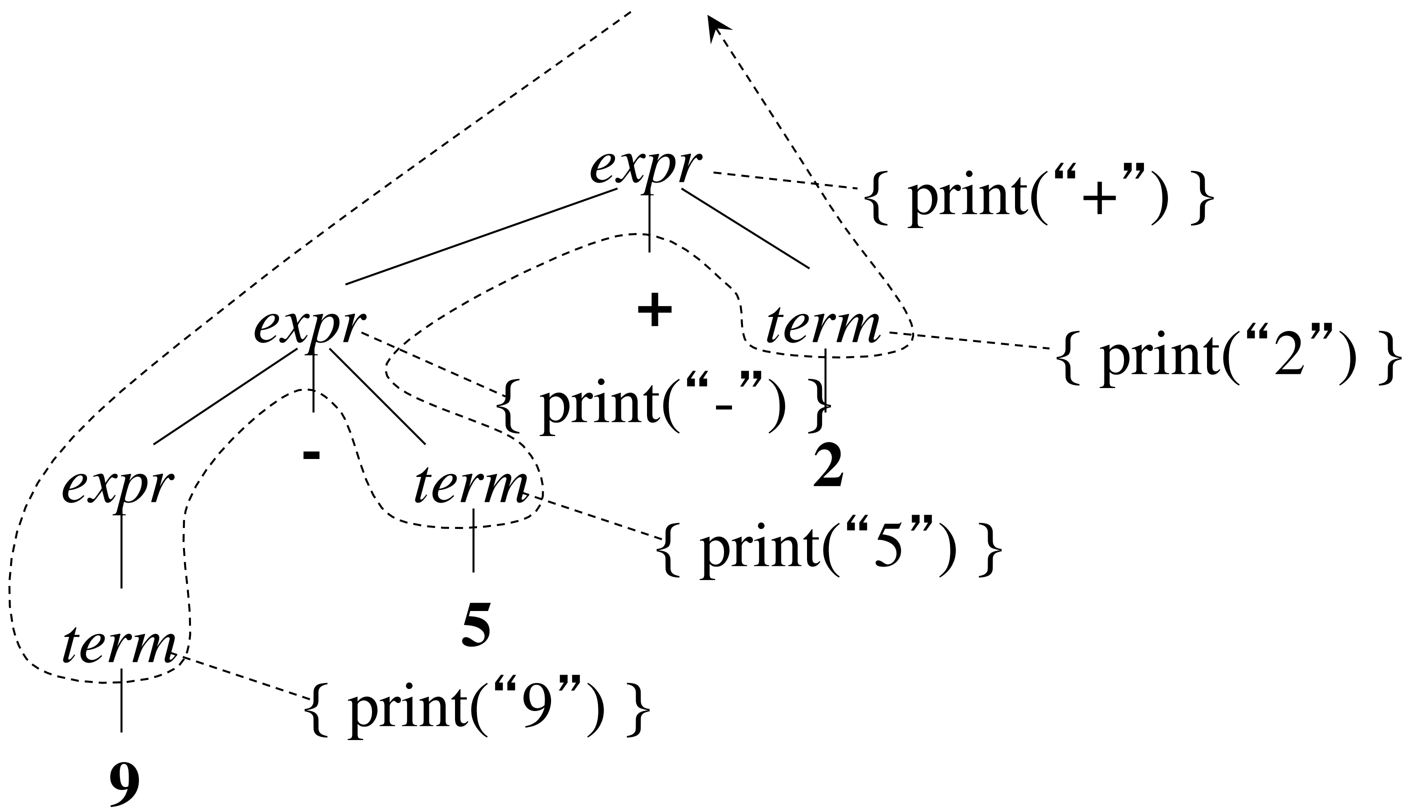
Embedded semantic action



Example Translation Scheme for Postfix Notation

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 source code
- *Top-down parsing* "constructs" a parse tree from root to leaves
- *Bottom-up parsing* "constructs" a parse tree from leaves to root

Predictive Parsing

- *Recursive descent parsing* is a top-down parsing method
  – Each nonterminal has one (recursive) procedure that is 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)

type()

Check *lookahead* and call *match*

match('**array**')

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

↑ *lookahead*

Example Predictive Parser (Execution Step 2)

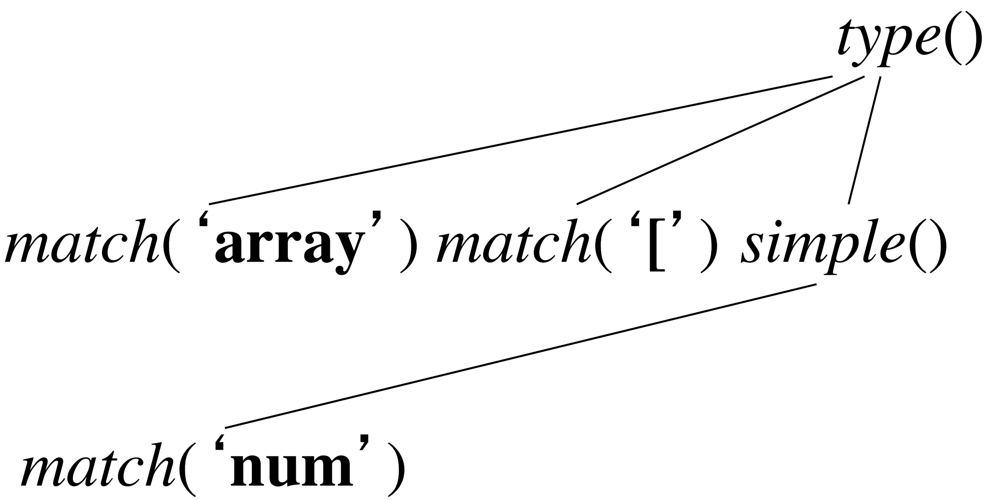
type()

match(‘**array**’) *match*(‘**[**’)

<u>Input</u>: **array** **[** **num** **dotdot** **num** **]** **of** **integer**

↑ *lookahead*

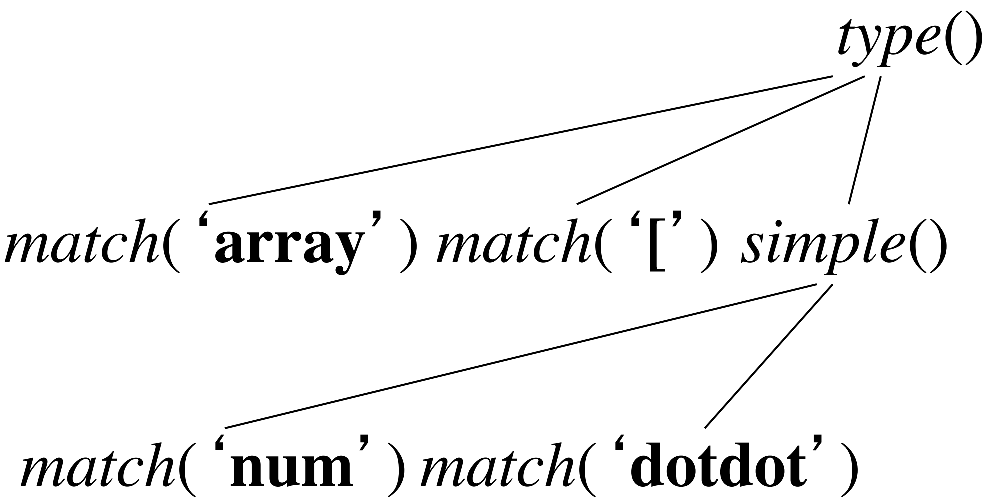
Example Predictive Parser (Execution Step 3)



<u>Input</u>: **array** **[** **num** **dotdot** **num** **]** **of** **integer**

↑ *lookahead*

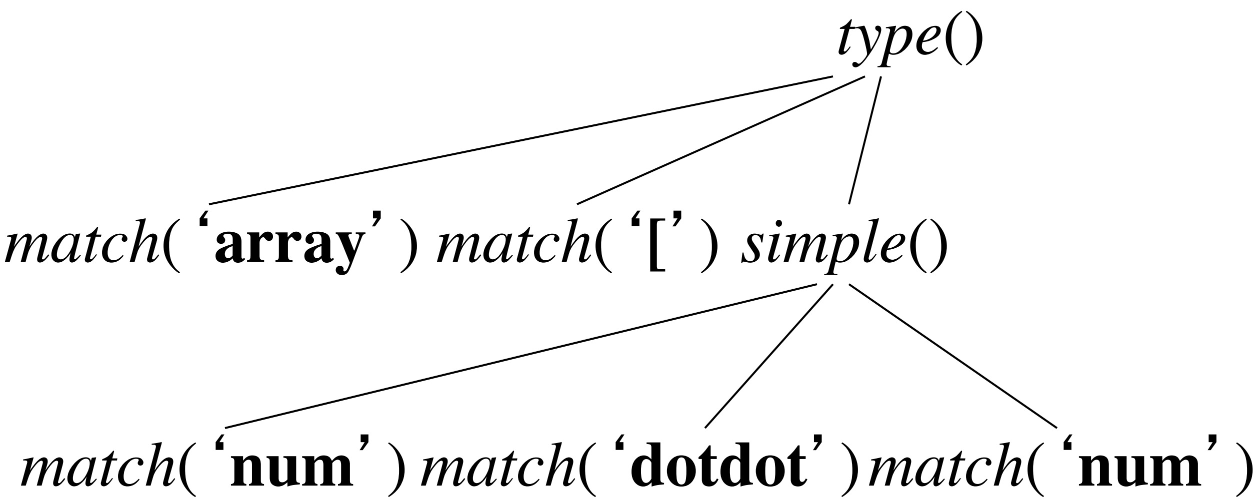
Example Predictive Parser (Execution Step 4)



<u>Input</u>: **array** **[** **num** **dotdot** **num** **]** **of** **integer**

↑ *lookahead*

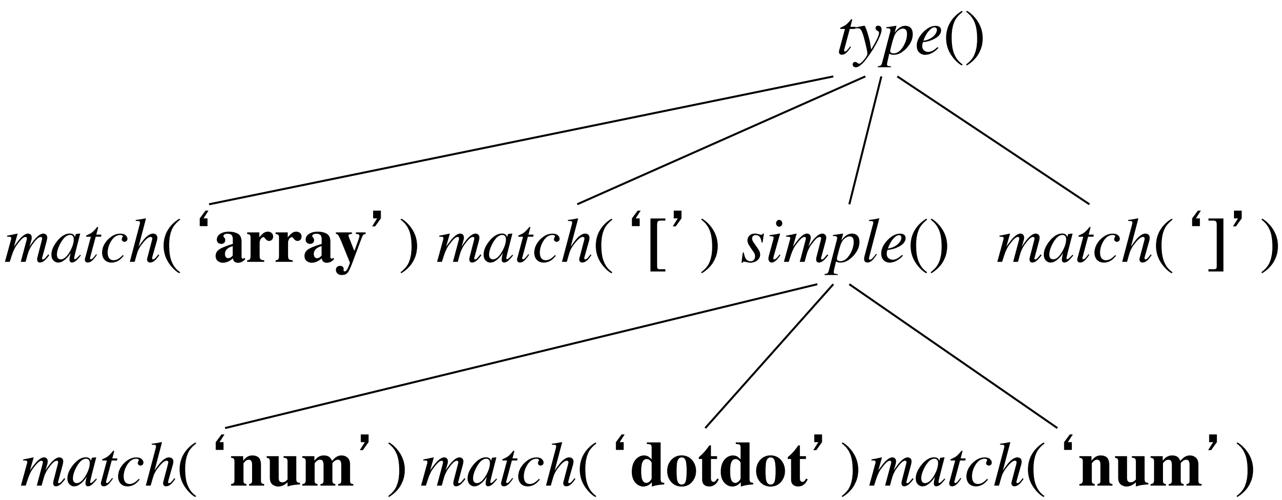
Example Predictive Parser (Execution Step 5)



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

↑ *lookahead*

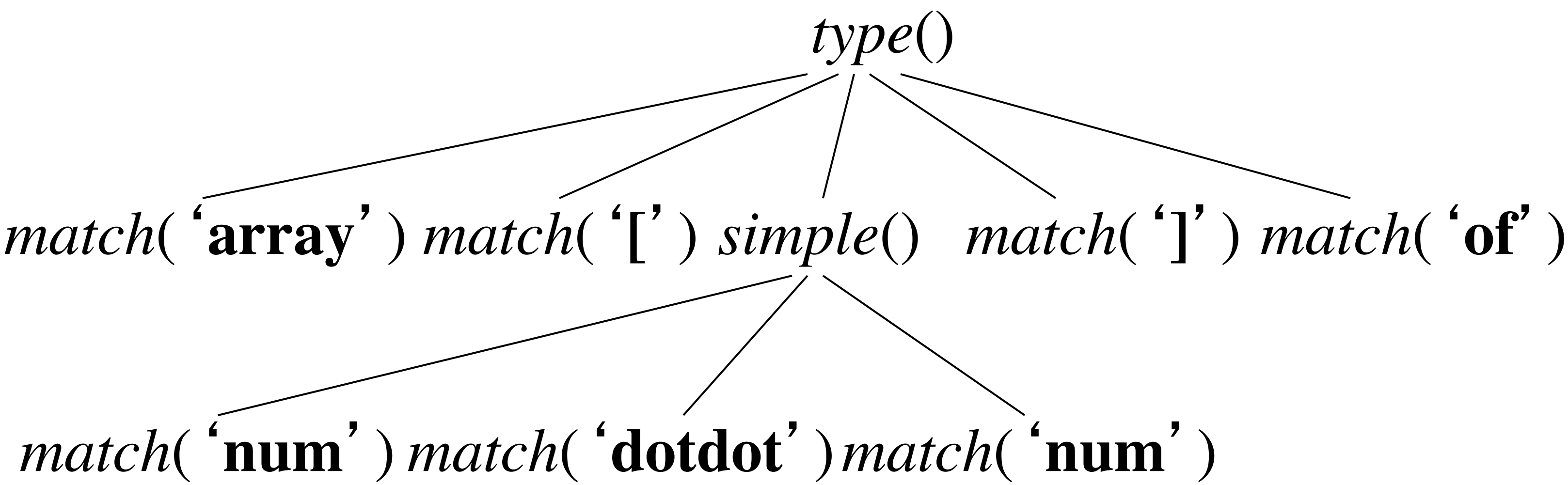
Example Predictive Parser (Execution Step 6)



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

↑ *lookahead*

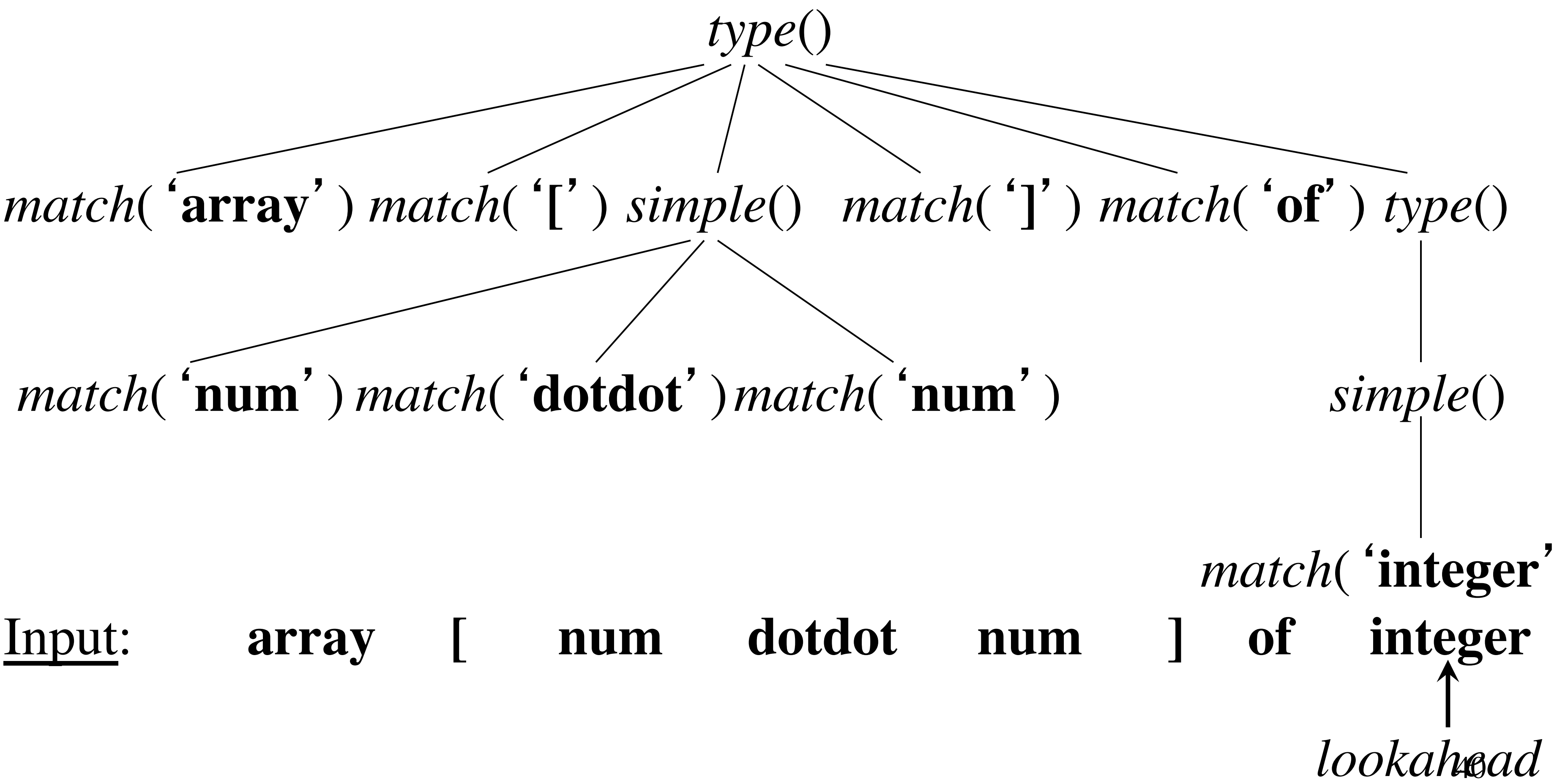
Example Predictive Parser (Execution Step 7)



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

↑ *lookahead*

Example Predictive Parser (Execution Step 8)



FIRST

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

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

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

How to use FIRST

We use FIRST to write a predictive parser as follows

expr → *term rest*
rest → **+** *term rest*
  | **-** *term rest*
  | ε

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

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

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* **endif**
| **if** expr **then** *stmt* **else** *stmt* **endif**

We can use *left factoring* to fix the problem

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

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

Code of the translator

$expr \rightarrow term\ rest$

$rest \rightarrow$ **+** $term$ { print("+") } $rest$

$rest \rightarrow$ **-** $term$ { print("-") } $rest$

$rest \rightarrow \varepsilon$

$term \rightarrow$ **0** { print("0") }

$term \rightarrow$ **1** { print("1") }

…

$term \rightarrow$ **9** { print("9") }

```
main()
{   lookahead = getchar();
    expr();
}
expr()
{   term(); rest(); }
rest ()
{ if (lookahead == '+')
        {match('+'); term(); putchar('+'); rest();
        }
  else if (lookahead == '-')
        {match('-'); term(); putchar('-'); rest();
        }
  else {};
}
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);
}
```

Optimized code of the translator

expr → *term rest*

rest → **+** *term* { print("+") } *rest*
rest → **-** *term* { print("-") } *rest*
rest → ε

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

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