Principles of Programming Languages

http://www.di.unipi.it/~andrea/Didattica/PLP-15/

Prof. Andrea Corradini
Department of Computer Science, Pisa

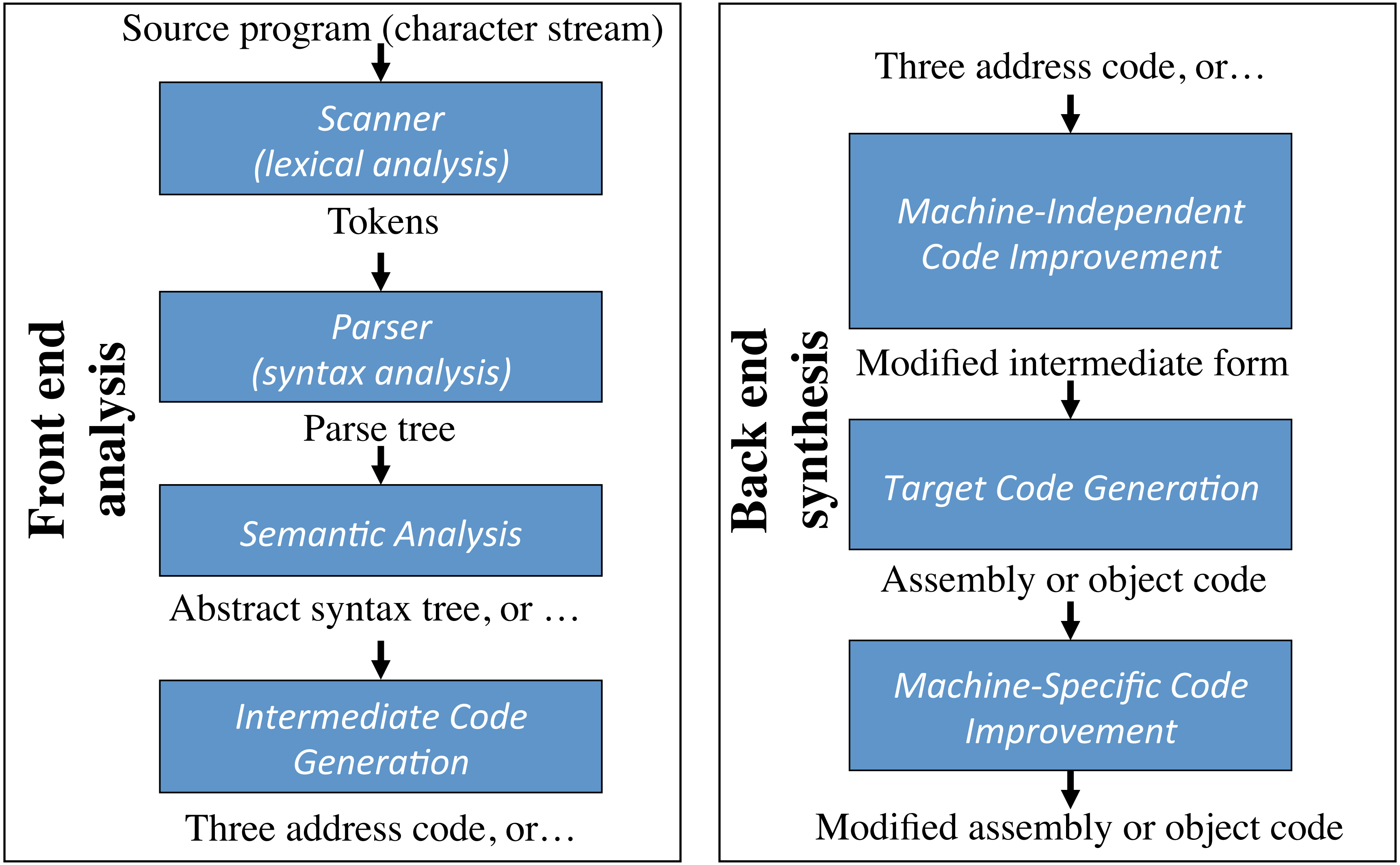
Lesson 3

- Overview of a syntax-directed compiler front-end

Overview of syntax-directed front-end

- (Context-Free) Grammars, Chomsky hierarchy
- Parse trees
- Ambiguity, associativity and precedence
- Syntax-directed translation
- Translation schemes
- Predictive recursive descent parsing
- Left factoring, elimination of left recursion

Compiler Front- and Back-end



Front end
analysis

Source program (character stream)

Scanner (lexical analysis)

Tokens

Parser (syntax analysis)

Parse tree

Semantic Analysis

Abstract syntax tree, or …

Intermediate Code Generation

Three address code, or…

Back end
synthesis

Three address code, or…

Machine-Independent Code Improvement

Modified intermediate form

Target Code Generation

Assembly or object code

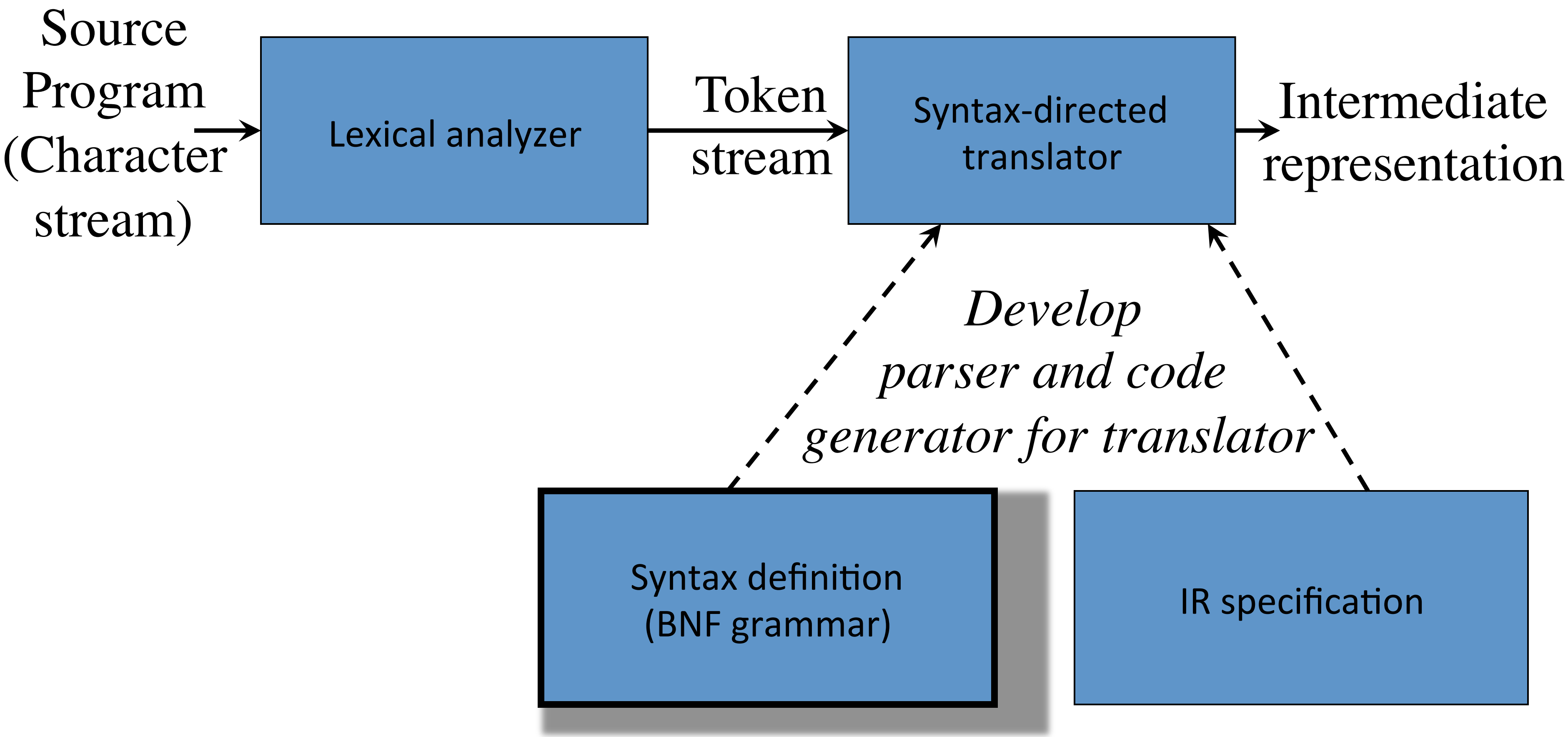
Machine-Specific Code Improvement

Modified assembly or object code

A simple syntax-directed Compiler Front-end

- Overview of the front-end of a compiler with:
  - Definition of the context-free syntax of a programming language
  - Presentation of a source code parser: *top-down predictive parsing*
  - *Lexical analysis*
  - Implementing *syntax directed translation* to generate intermediate code

The Structure of the Front-End



Source Program (Character stream)

Lexical analyzer

Token stream

Syntax-directed translator

Intermediate representation

Develop parser and code generator for translator

Syntax definition (BNF grammar)

IR specification

Syntax Definition: Grammars

- A **grammar** is a 4-tuple G = (N, T, P, S) where
  - T is a finite set of tokens (*terminal* symbols)
  - N is a finite set of *nonterminals*
  - P is a finite set of *productions* of the form
    $$\alpha \rightarrow \beta$$
    where $\alpha \in (N \cup T)^* \, N \, (N \cup T)^*$ and $\beta \in (N \cup T)^*$
  - $S \in N$ is a designated *start symbol*
- **A*** is the set of finite sequences of elements of **A**. If **A** = {a,b}, **A*** = {ε, a, b, aa, ab, ba, bb, aaa, …}
- **AB** = {ab | a $\in$ **A**, b $\in$ **B**}

Notational Conventions Used

- Terminals
  $a,b,c,\ldots \in T$
  specific terminals: **0**, **1**, **id**, **+**
- Nonterminals
  $A,B,C,\ldots \in N$
  specific nonterminals: *expr*, *term*, *stmt*
- Grammar symbols
  $X,Y,Z \in (N \cup T)$
- Strings of terminals
  $u,v,w,x,y,z \in T^*$
- Strings of grammar symbols
  $\alpha,\beta,\gamma \in (N \cup T)^*$

Derivations

- A *one-step derivation* is defined by
  $$\gamma \alpha \delta \Rightarrow \gamma \beta \delta$$
  where $\alpha \rightarrow \beta$ is a production in the grammar
- In addition, we define
  - $\Rightarrow$ is *leftmost* $\Rightarrow_{lm}$ if γ does not contain a nonterminal
  - $\Rightarrow$ is *rightmost* $\Rightarrow_{rm}$ if δ does not contain a nonterminal
  - Transitive closure $\Rightarrow^*$ (zero or more steps)
  - Positive closure $\Rightarrow^+$ (one or more steps)
- α is a *sentential form* if $S \Rightarrow^* \alpha$
- The *language generated by G* is defined by

$$L(G) = \{w \in T^* \mid S \Rightarrow^+ w\}$$

Derivation (Example)

Grammar $G = (\{E\}, \{\mathbf{+},\mathbf{*},\mathbf{(},\mathbf{)},\mathbf{-},\mathbf{id}\}, P, E)$ with productions

$P \; = \quad E \rightarrow E \mathbf{+} E$

$E \rightarrow E \mathbf{*} E$

$E \rightarrow \mathbf{(}\ E\ \mathbf{)}$

$E \rightarrow \mathbf{-}\ E$

$E \rightarrow \mathbf{id}$

Example derivations:

$$E \Rightarrow \mathbf{-}\ E \Rightarrow \mathbf{-\ id}$$

$$E \Rightarrow_{rm} E \mathbf{+} E \Rightarrow_{rm} E \mathbf{+ id} \Rightarrow_{rm} \mathbf{id + id}$$

$$E \Rightarrow^{*} E$$

$$E \Rightarrow^{*} \mathbf{id + id}$$

$$E \Rightarrow^{+} \mathbf{id * id + id}$$

Another grammar for expressions

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

Productions $P =$
$\textit{list} \rightarrow \textit{list}\ \mathbf{+}\ \textit{digit}$
$\textit{list} \rightarrow \textit{list}\ \mathbf{-}\ \textit{digit}$
$\textit{list} \rightarrow \textit{digit}$
$\textit{digit} \rightarrow \mathbf{0} \mid \mathbf{1} \mid \mathbf{2} \mid \mathbf{3} \mid \mathbf{4} \mid \mathbf{5} \mid \mathbf{6} \mid \mathbf{7} \mid \mathbf{8} \mid \mathbf{9}$

A *leftmost derivation*:

$$
\begin{array}{l}
\underline{\textit{list}} \\
\Rightarrow_{lm} \underline{\textit{list}}\ \mathbf{+}\ \textit{digit} \\
\Rightarrow_{lm} \underline{\textit{list}}\ \mathbf{-}\ \textit{digit}\ \mathbf{+}\ \textit{digit} \\
\Rightarrow_{lm} \underline{\textit{digit}}\ \mathbf{-}\ \textit{digit}\ \mathbf{+}\ \textit{digit} \\
\Rightarrow_{lm} \mathbf{9}\ \mathbf{-}\ \underline{\textit{digit}}\ \mathbf{+}\ \textit{digit} \\
\Rightarrow_{lm} \mathbf{9}\ \mathbf{-}\ \mathbf{5}\ \mathbf{+}\ \underline{\textit{digit}} \\
\Rightarrow_{lm} \mathbf{9}\ \mathbf{-}\ \mathbf{5}\ \mathbf{+}\ \mathbf{2}
\end{array}
$$

Chomsky Hierarchy: Language Classification

- A grammar *G* is said to be
  - *Regular* if it is *right linear* where each production is of the form
    $A \rightarrow w\,B$ or $A \rightarrow w$
    or *left linear* where each production is of the form
    $A \rightarrow B\,w$ or $A \rightarrow w$ $(w \in T^*)$
  - *Context free* if each production is of the form
    $A \rightarrow \alpha$
    where $A \in N$ and $\alpha \in (N \cup T)^*$
  - *Context sensitive* if each production is of the form
    $\alpha\,A\,\beta \rightarrow \alpha\,\gamma\,\beta$
    where $A \in N$, $\alpha,\gamma,\beta \in (N \cup T)^*$, $|\gamma| > 0$
  - *Unrestricted*

Chomsky Hierarchy

$$\mathcal{L}(\mathit{regular}) \subset \mathcal{L}(\mathit{context\ free}) \subset \mathcal{L}(\mathit{context\ sensitive}) \subset \mathcal{L}(\mathit{unrestricted})$$

Where $\mathcal{L}(T) = \{\ L(G) \mid G \text{ is of type } T\ \}$
That is: the set of all languages
generated by grammars G of type T

Examples:

Every *finite language* is regular! (construct a FSA for strings in $L(G)$)

$L_1 = \{\ \mathbf{a}^n\mathbf{b}^n \mid n \geq 1\ \}$ is context free

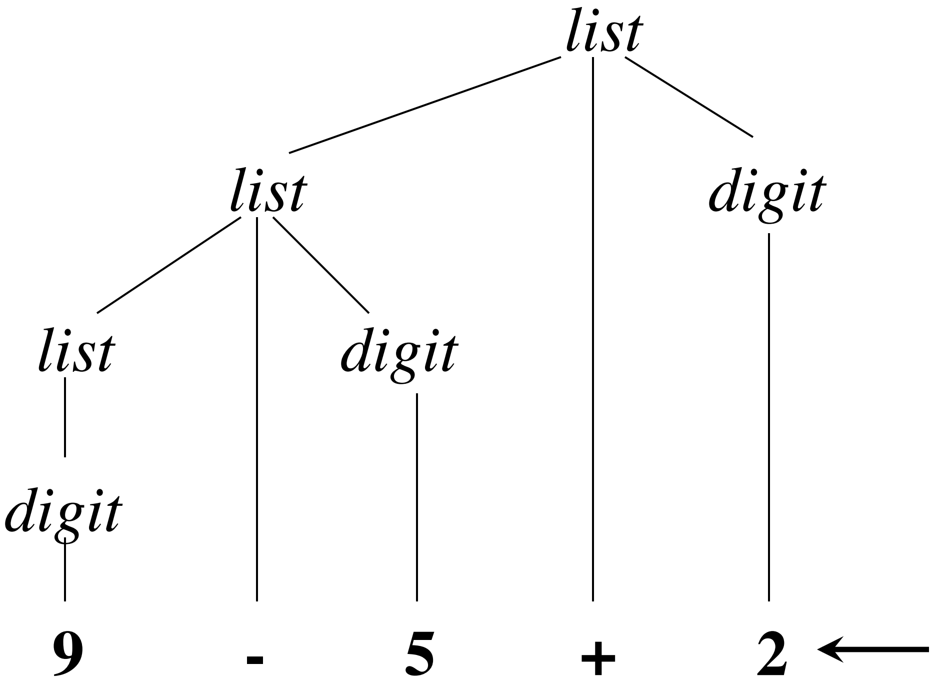
$L_2 = \{\ \mathbf{a}^n\mathbf{b}^n\mathbf{c}^n \mid n \geq 1\ \}$ is context sensitive

Parse Trees (context-free grammars)

- Tree-shaped representation of derivations
- 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 *internal 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:

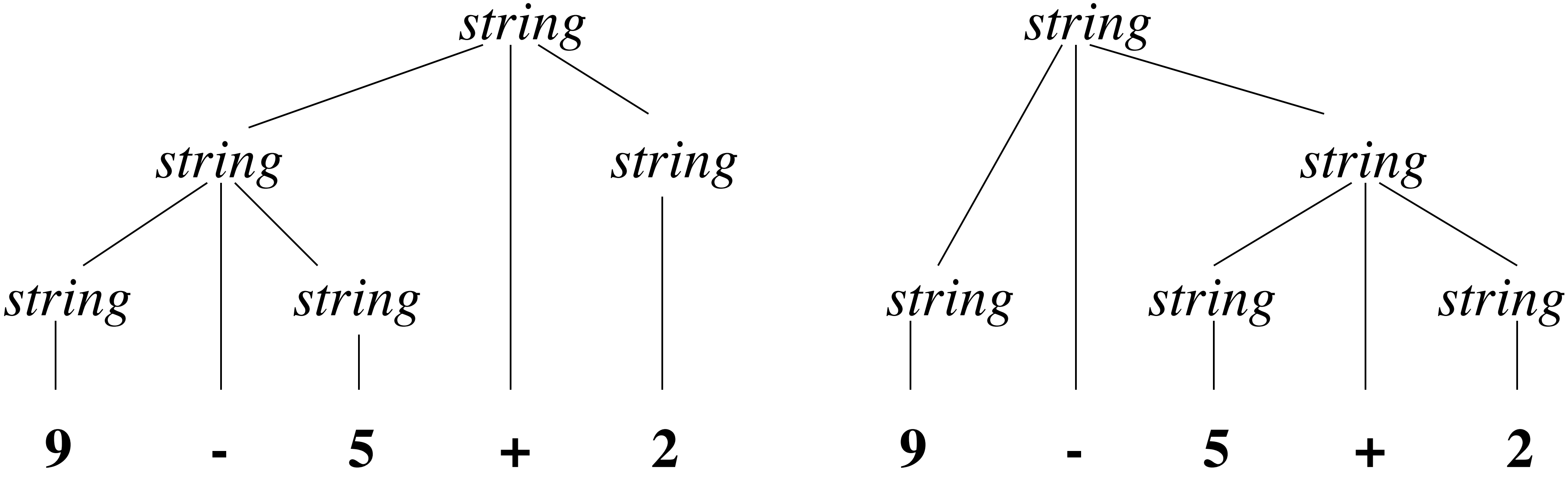
$$G = <\{string\}, \{\mathbf{+},\mathbf{-},\mathbf{0},\mathbf{1},\mathbf{2},\mathbf{3},\mathbf{4},\mathbf{5},\mathbf{6},\mathbf{7},\mathbf{8},\mathbf{9}\}, P, string>$$

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 → *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*

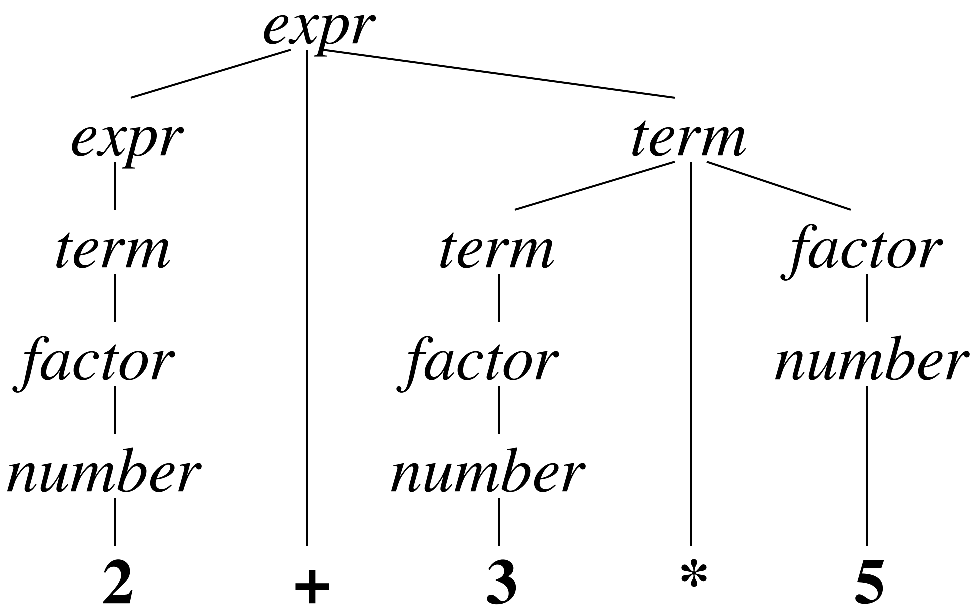
String **a=b=c** has the same meaning as **a=(b=c)**

Precedence of Operators

Operators with higher precedence “bind more tightly”

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

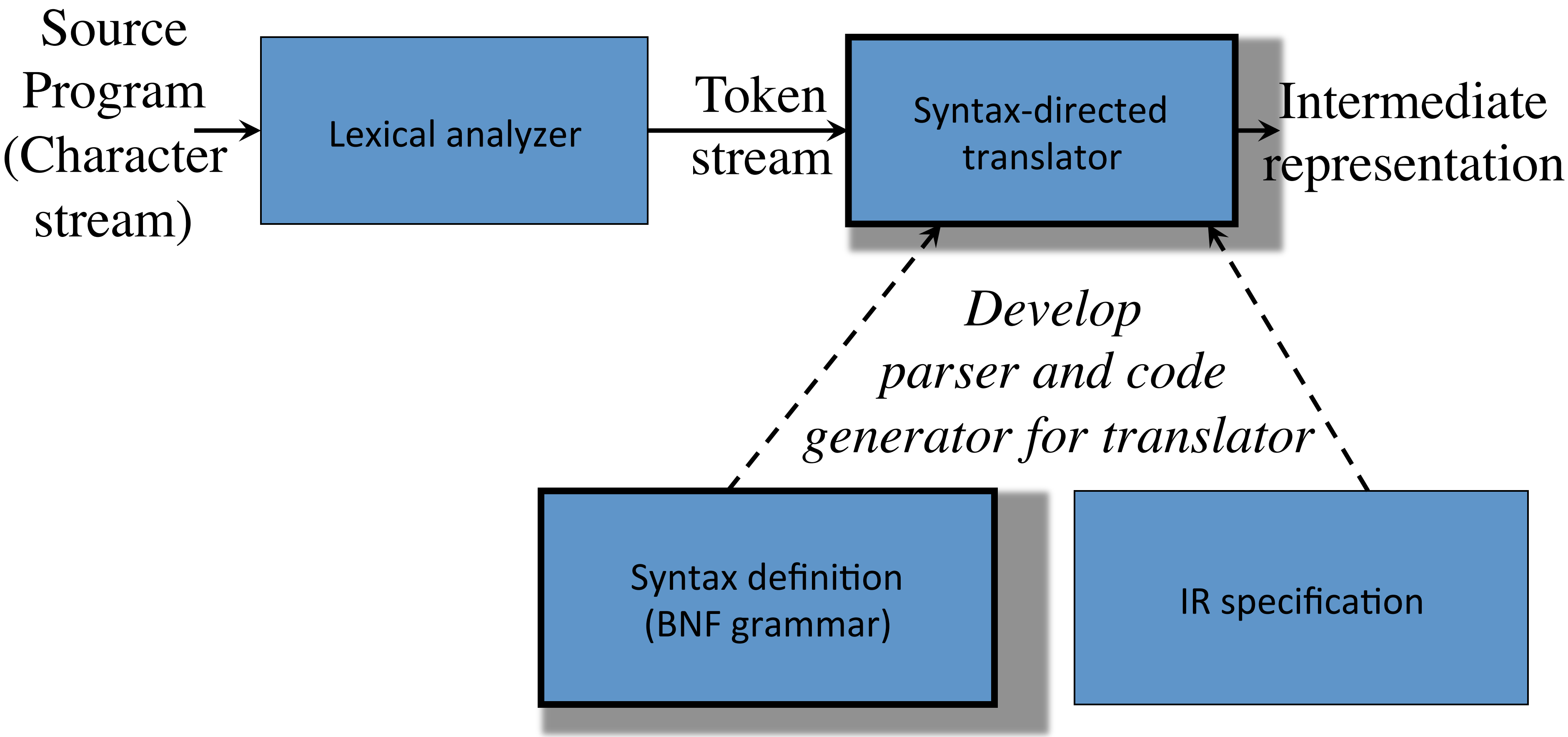
String **2+3*5** has the same meaning as **2+(3*5)**



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

The Structure of the Front-End



Source Program (Character stream)

Lexical analyzer

Token stream

Syntax-directed translator

Intermediate representation

Develop parser and code generator for translator

Syntax definition (BNF grammar)

IR specification

Syntax-Directed Translation

- Uses a Context Free 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

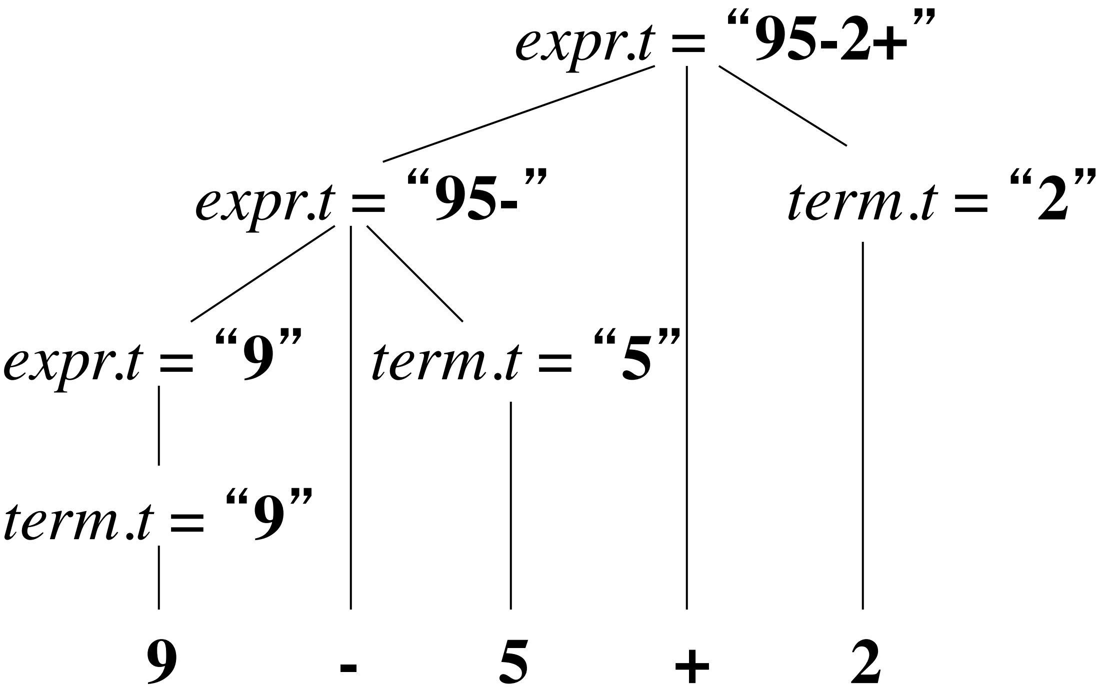
- An attribute is said to be …
  - *synthesized* if its value at a parse-tree node is determined from the attribute values at the children of the node
  - *inherited* if its value at a parse-tree node is determined by the parent (by enforcing the parent's semantic rules)

Example Attribute Grammar (Postfix Form)

String concat operator

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

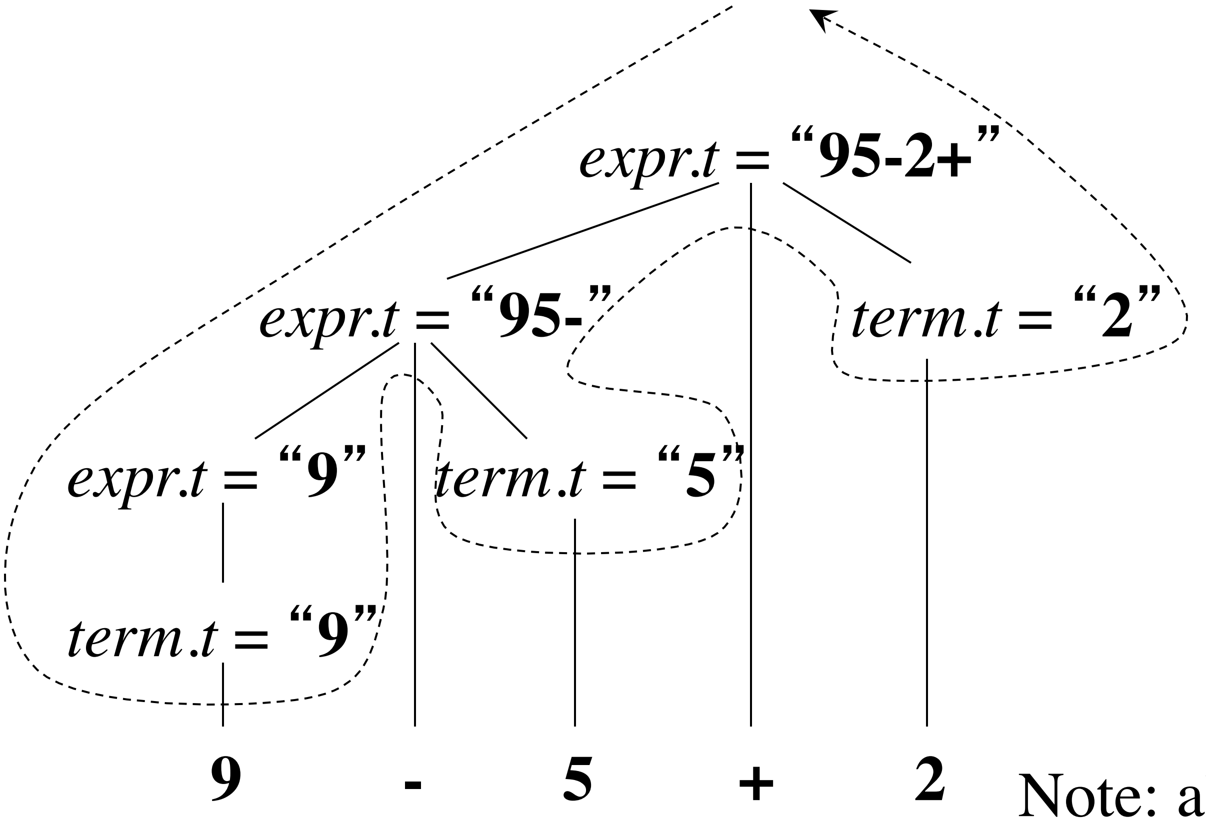
Example Annotated Parse Tree



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)

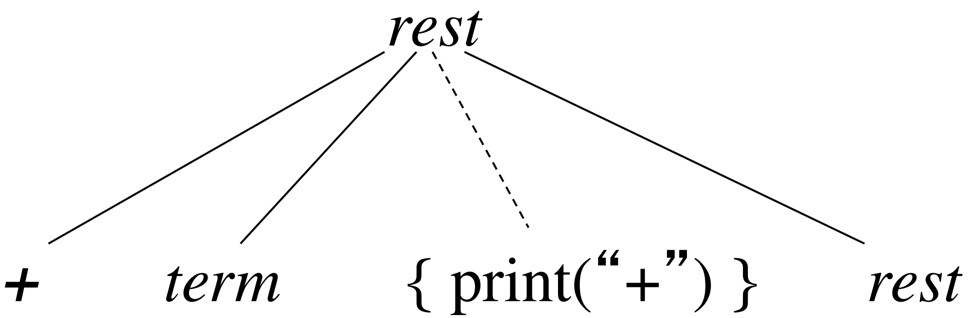


Note: all attributes are of the synthesized type

Translation Schemes

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

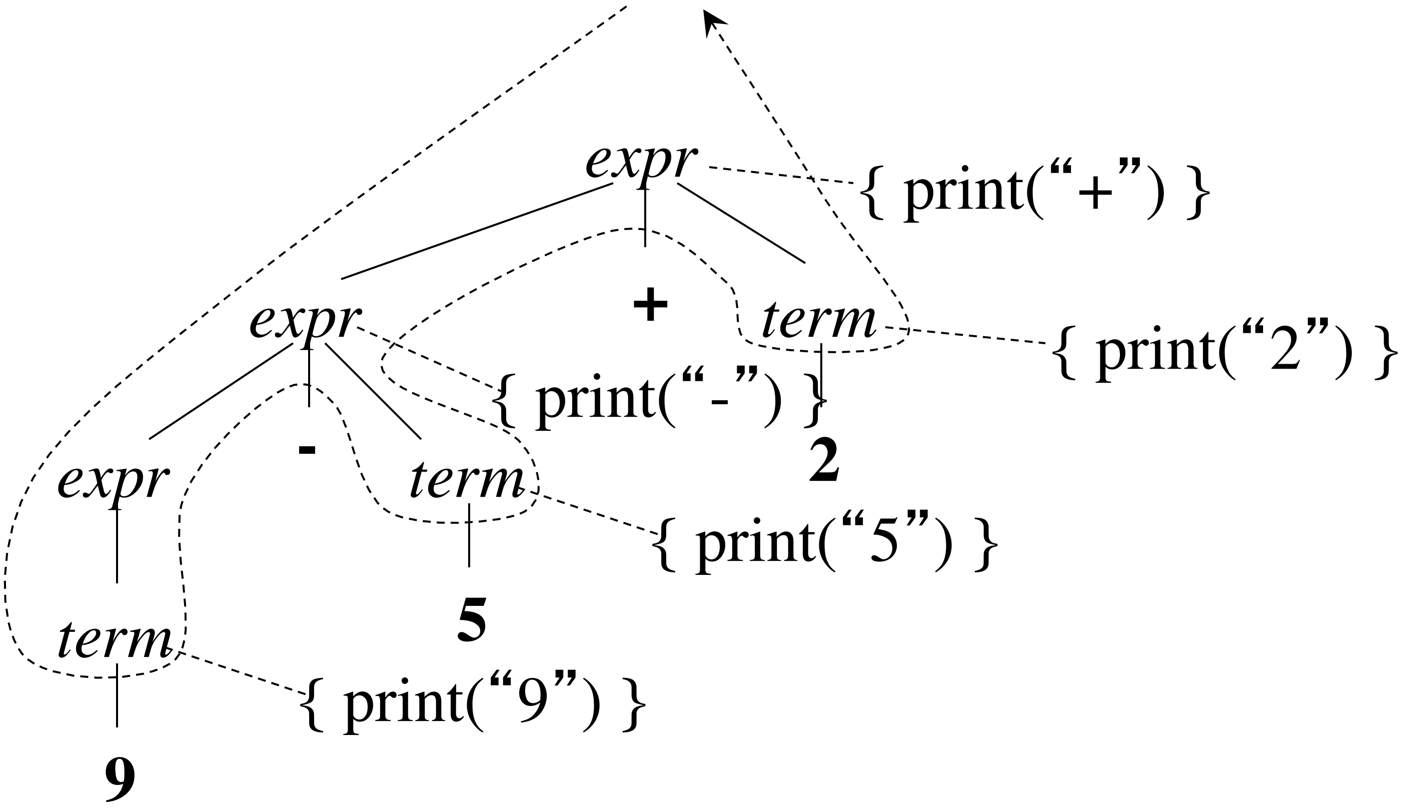
$$rest \to \mathbf{+}\ term\ \underbrace{\{\ \mathrm{print}(\text{“+”})\ \}}_{\text{Embedded semantic action}}\ rest$$



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

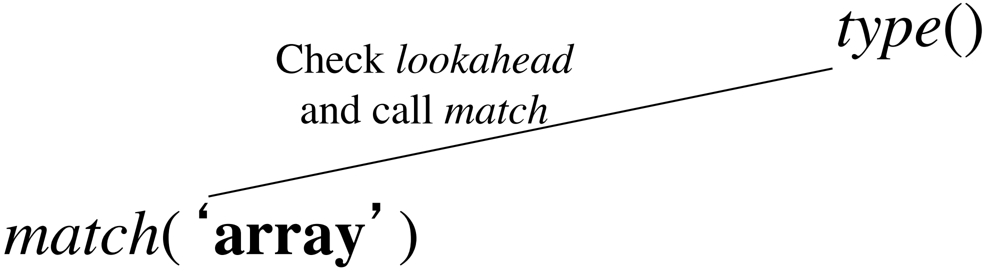
type → *simple*
| **^ id**
| **array [** *simple* **] of** *type*
simple → **integer**
| **char**
| **num dotdot num**

```
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 match(t : token);
begin
   if lookahead = t then
      lookahead := nexttoken()
   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;
```

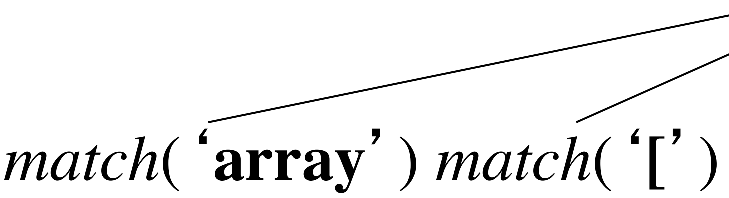
Example Predictive Parser (Execution Step 1)



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

↑ *lookahead*

Example Predictive Parser (Execution Step 2)



type()

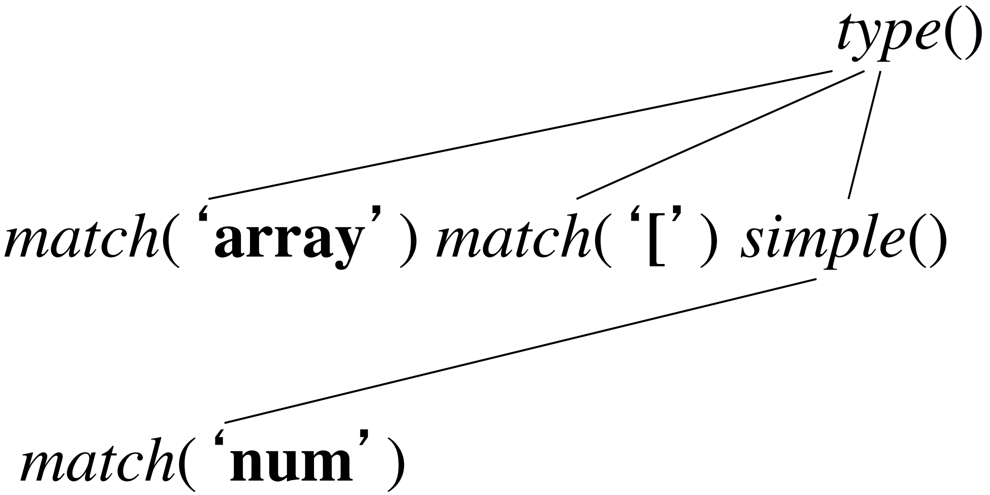
match(‘**array**’) *match*(‘**[**’)

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

↑

lookahead

Example Predictive Parser (Execution Step 3)



type()

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

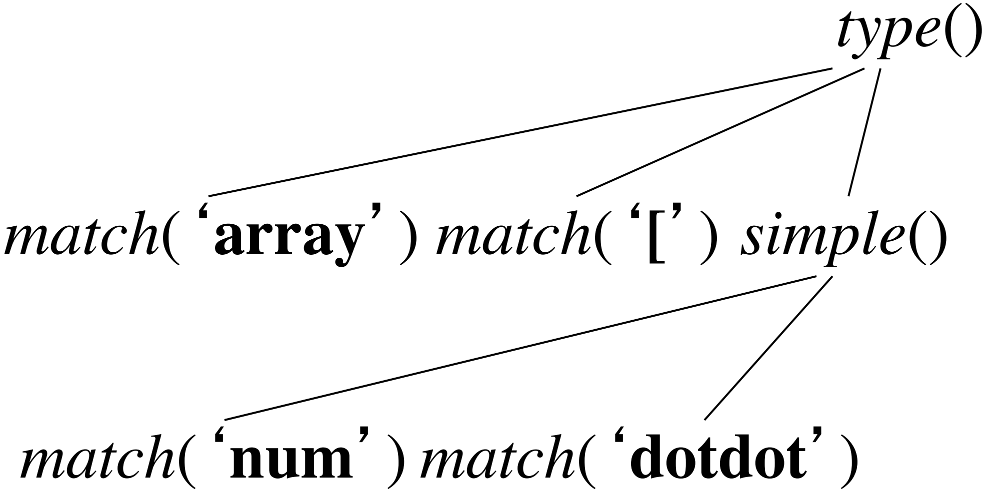
match(‘**num**’)

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

↑

lookahead

Example Predictive Parser (Execution Step 4)


type()

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

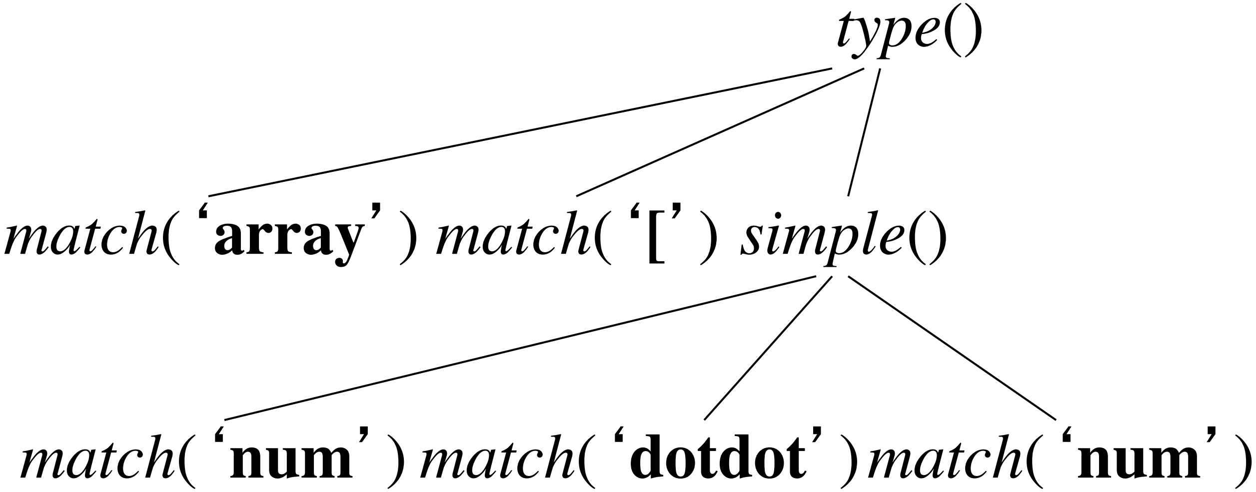
match(‘**num**’) *match*(‘**dotdot**’)

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

↑

lookahead

Example Predictive Parser (Execution Step 5)



type()

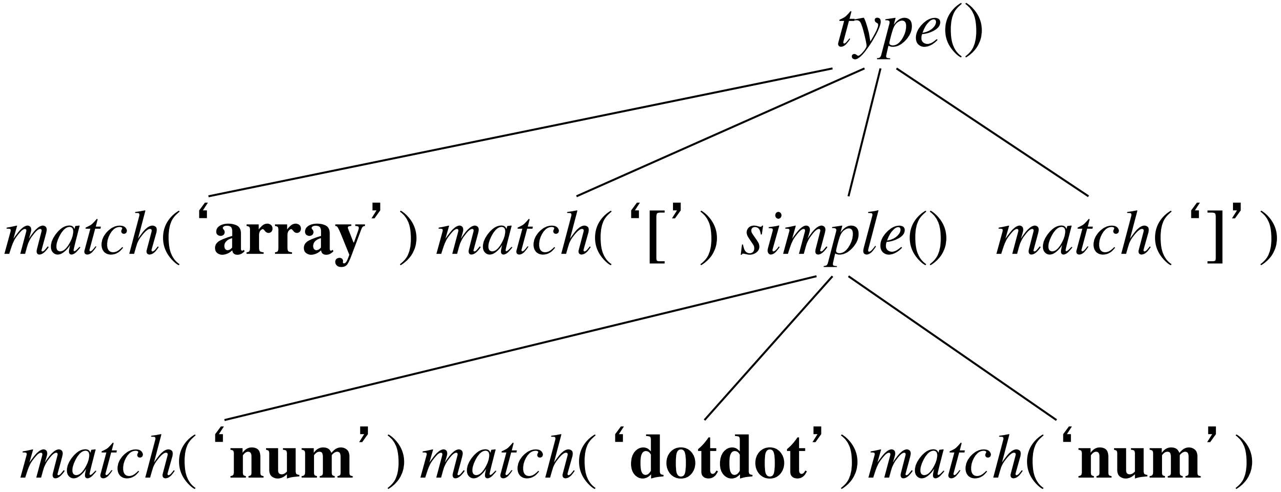
match(‘**array**’) *match*(‘**[**’) *simple*()

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

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

↑ *lookahead* (under the second **num**)

Example Predictive Parser (Execution Step 6)



type()

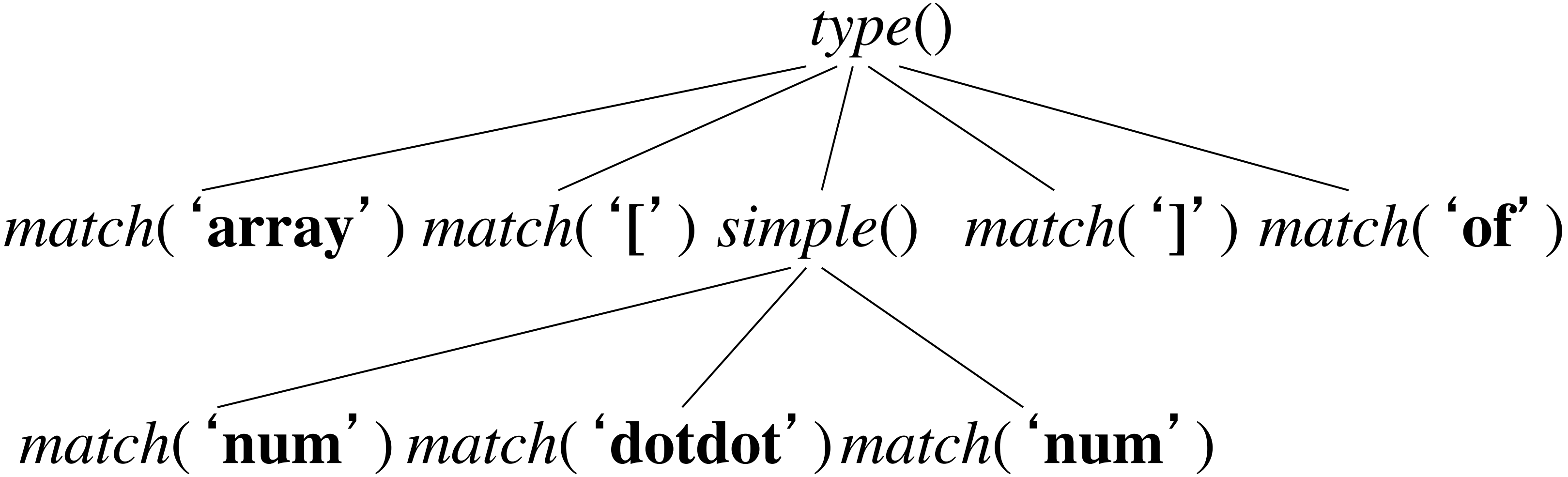
match(‘**array**’) *match*(‘[’) *simple*() *match*(‘]’)

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

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

lookahead

Example Predictive Parser (Execution Step 7)



type()

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

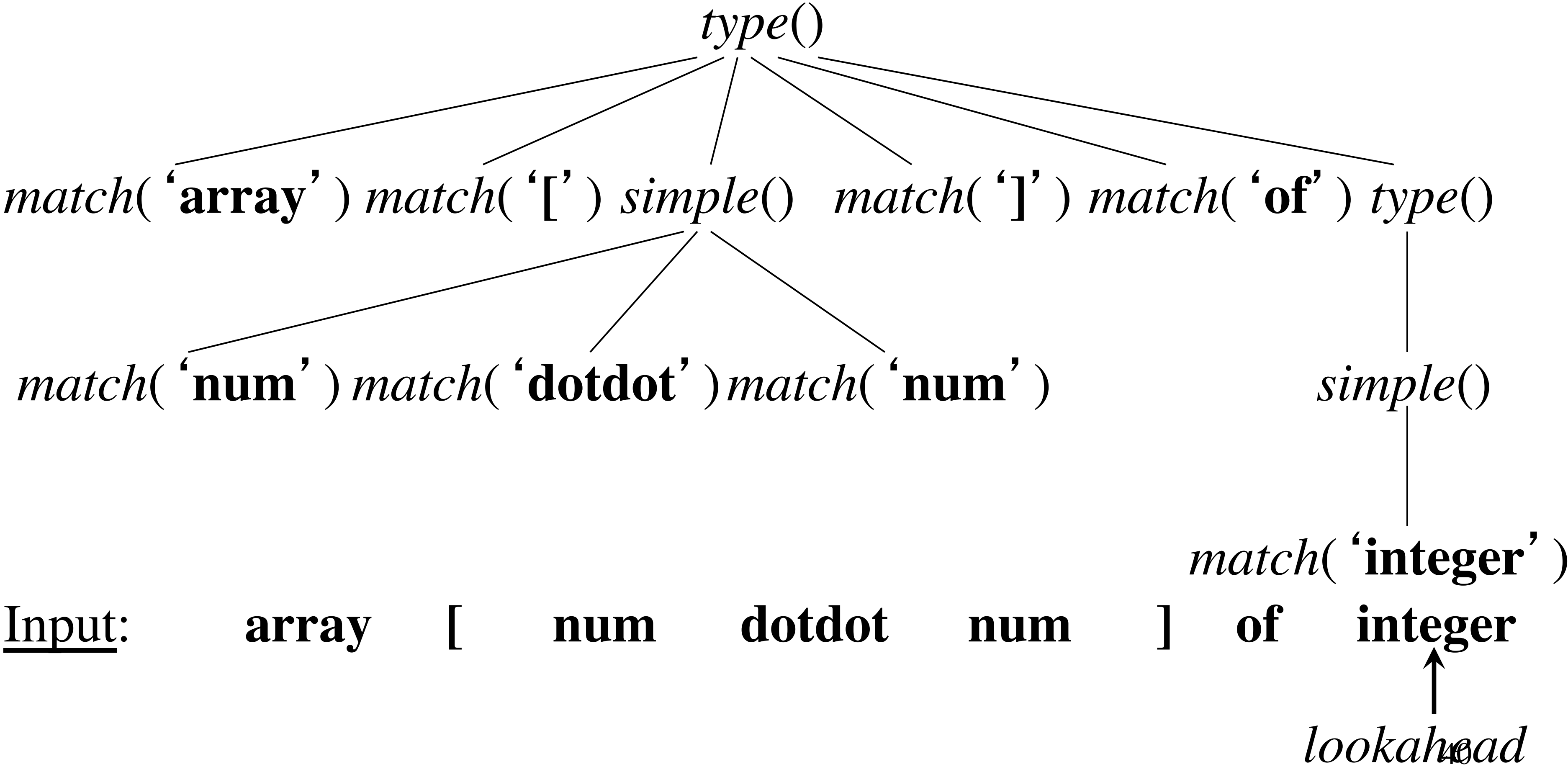
match(‘**num**’) *match*(‘**dotdot**’) *match*(‘**num**’)

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

↑

lookahead

Example Predictive Parser (Execution Step 8)



type()

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

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

match(**‘integer’**)

<u>Input</u>: **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

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

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

```
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* **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*

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