Principles of Programming Languages

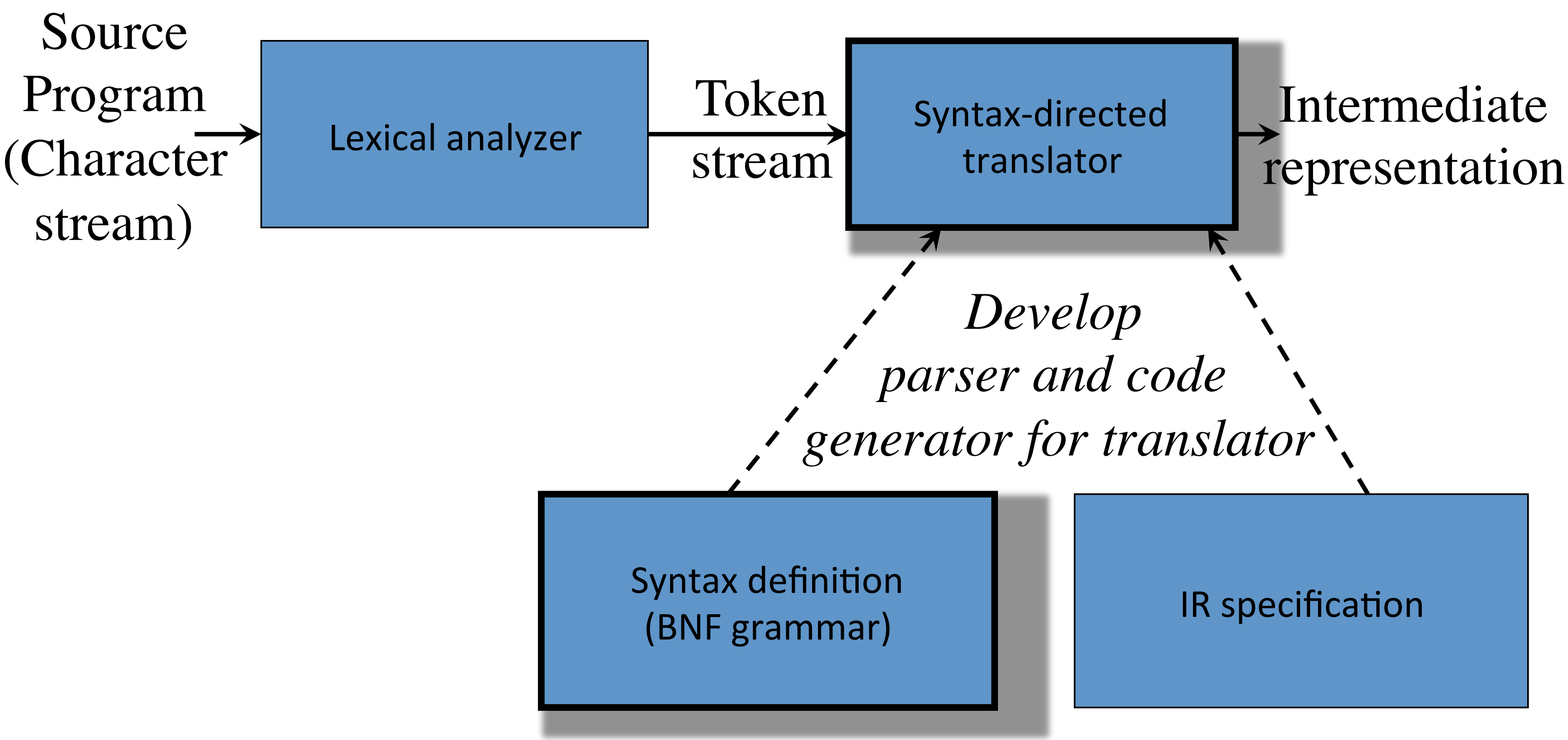
http://www.di.unipi.it/~andrea/Didattica/PLP-15/

Prof. Andrea Corradini
Department of Computer Science, Pisa

Lesson 4

- Overview of a Simple Compiler Front-end
  - Syntax-directed translation
  - Translation schemes
  - Lexical analysis
  - Left factoring, elimination of left recursion
  - Intermediate code generation
  - Static checking

The Structure of the Front-End



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
- The parse tree is traversed and semantic rules are applied: after the tree traversal is completed, the attribute values on nonterminals contain the translated form of the input
- Can be used for syntactic and static semantic analysis

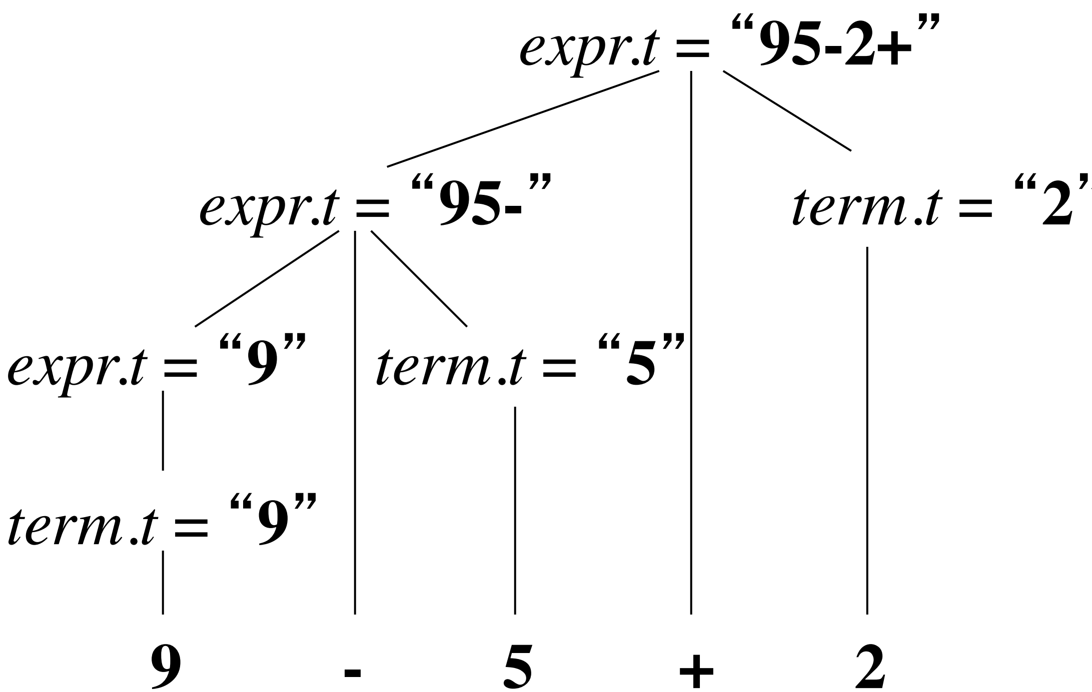
Example Attribute Grammar: Generating Postfix Form of Expressions

- Each symbol **X** of the grammar has one attribute "**t**" of type *string*, denoted **X.t**

String concat operator

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

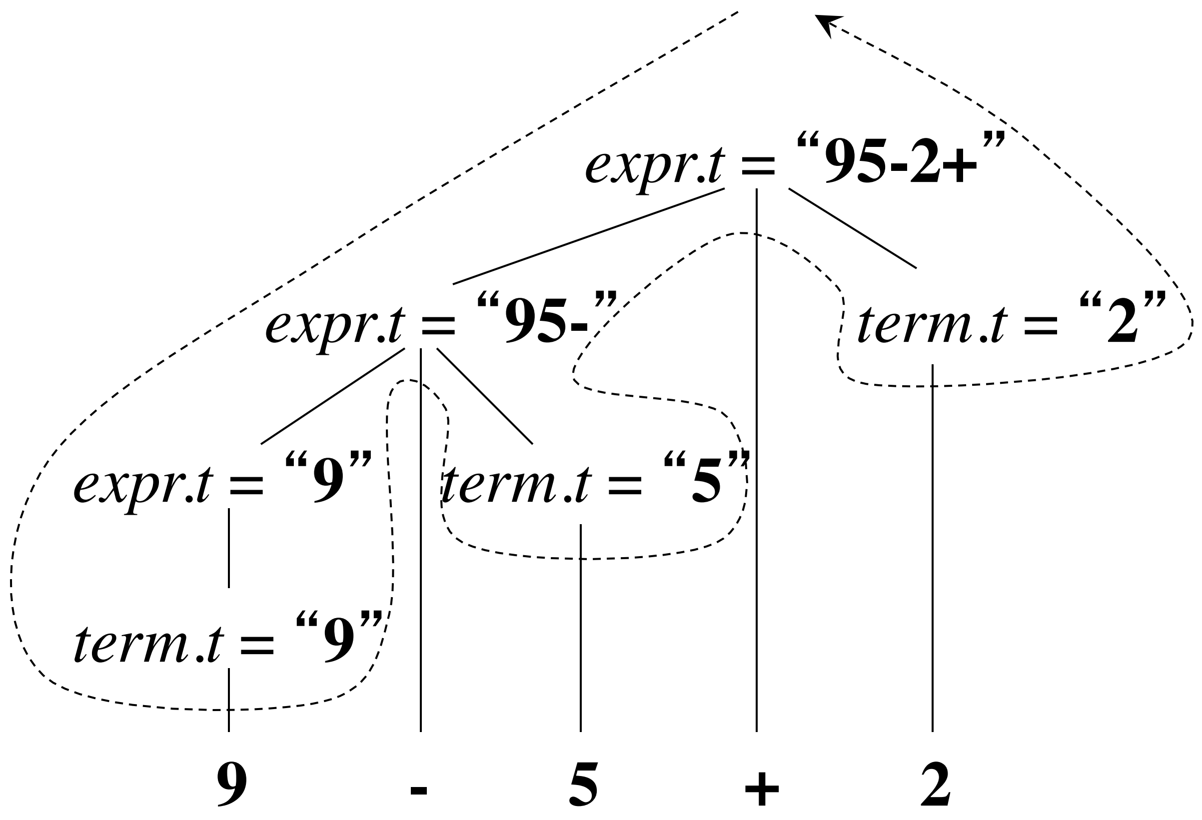
Example Annotated Parse Tree



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)



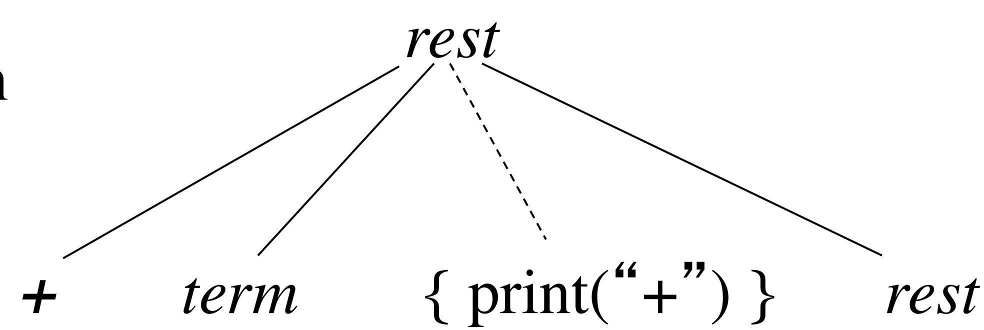
Synthesized and Inherited Attributes

- 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)
- The previous example only uses *synthesized attributes*
- If inherited attributes are present, depth-first traversal could not work

Translation Schemes

- A *translation scheme* is a CF grammar embedded with *semantic actions*
- Semantic actions are seen as terminal symbols in the parse tree, and they are executed during a depth-first, left-to-right traversal

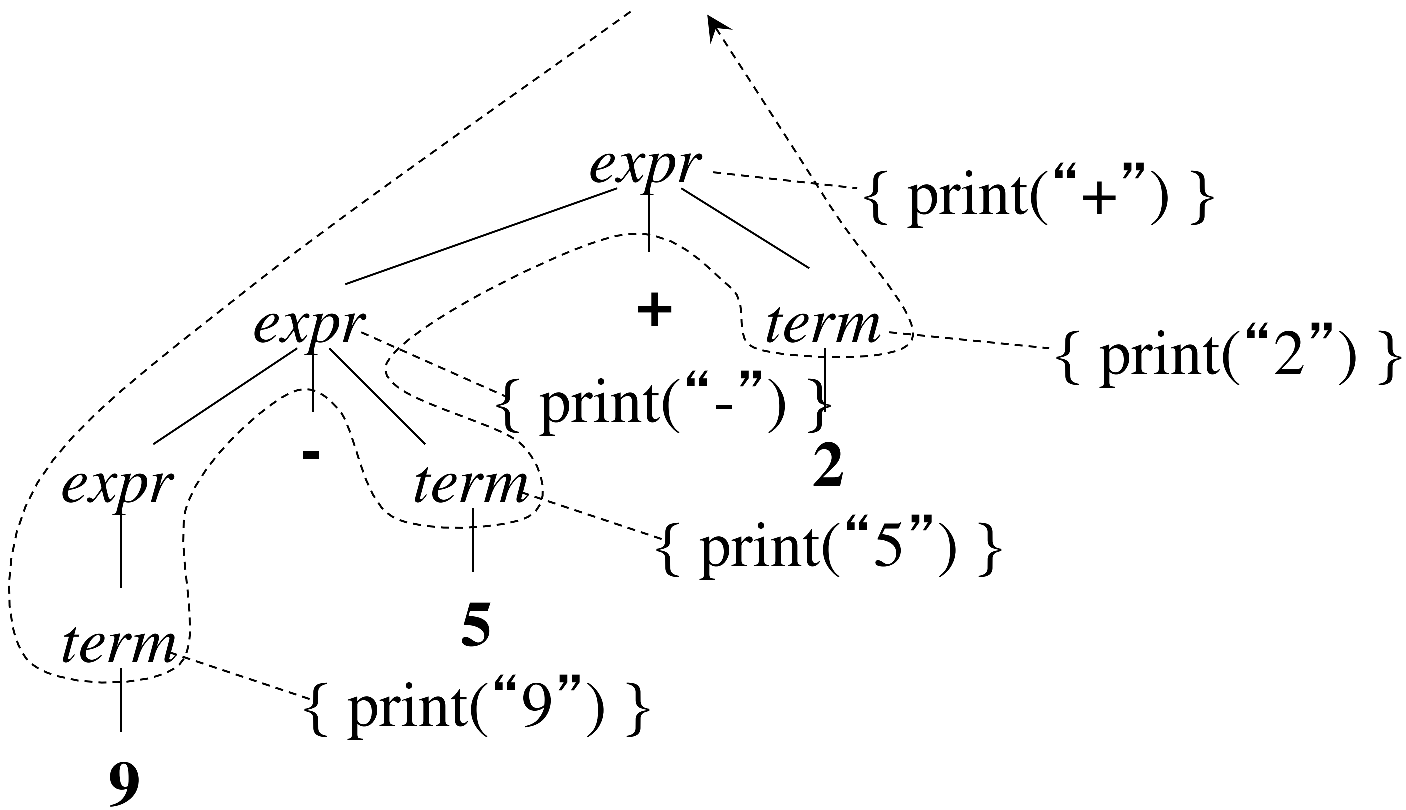
$$rest \rightarrow \mathbf{+}\ term\ \underbrace{\{\ \mathrm{print}(\text{“+”})\ \}}_{\text{Embedded semantic action}}\ rest$$



Example Translation Scheme for Postfix Notation

$expr \rightarrow expr$ **+** $term$ { print("+") }
$expr \rightarrow expr$ **-** $term$ { print("-") }
$expr \rightarrow term$
$term \rightarrow$ **0** { print("0") }
$term \rightarrow$ **1** { print("1") }
… …
$term \rightarrow$ **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, they are tried in sequence till one succeeds. If all fail, the procedure for the non-terminal fails.
  - Backtracking is necessary, and complexity is exponential in genera
- ***Predictive parsing*** is a special form of recursive descent parsing where one uses lookahead tokens to unambiguously determine the production to try. Complexity is linear.

Example Predictive Parser

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

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)

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)

type()

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

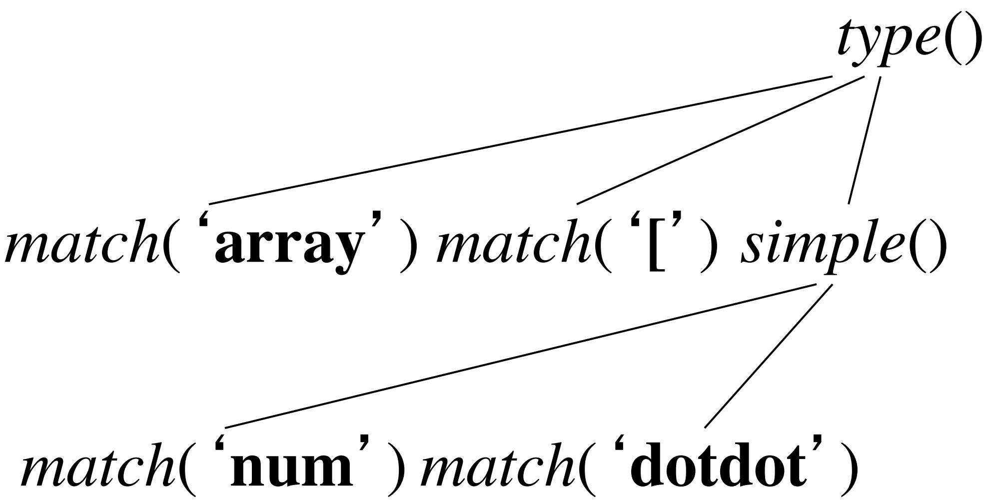
match(‘**num**’)

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

↑

lookahead

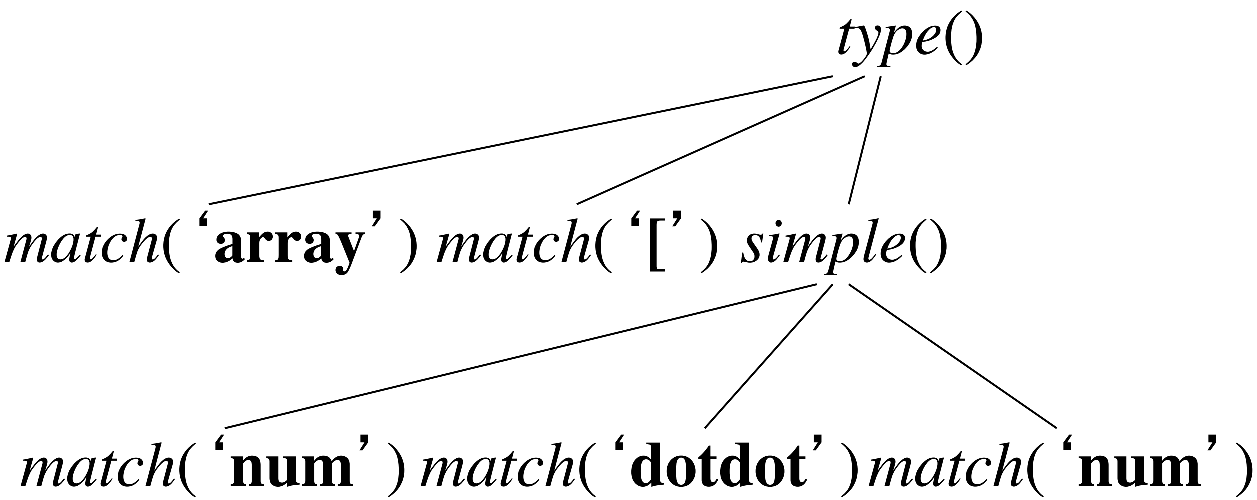
Example Predictive Parser (Execution Step 4)



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

↑
lookahead

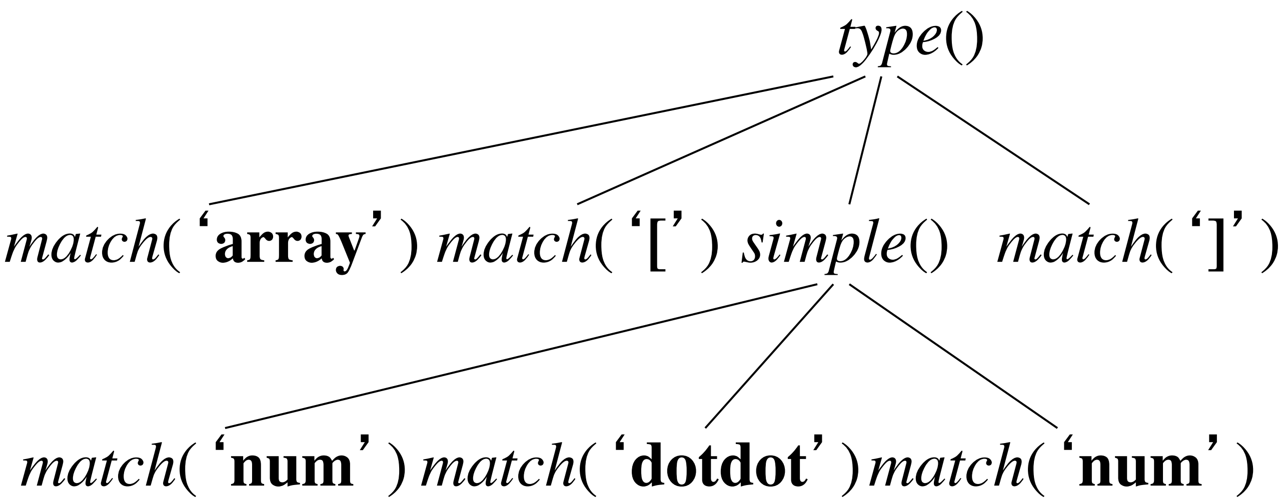
Example Predictive Parser (Execution Step 5)



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

lookahead

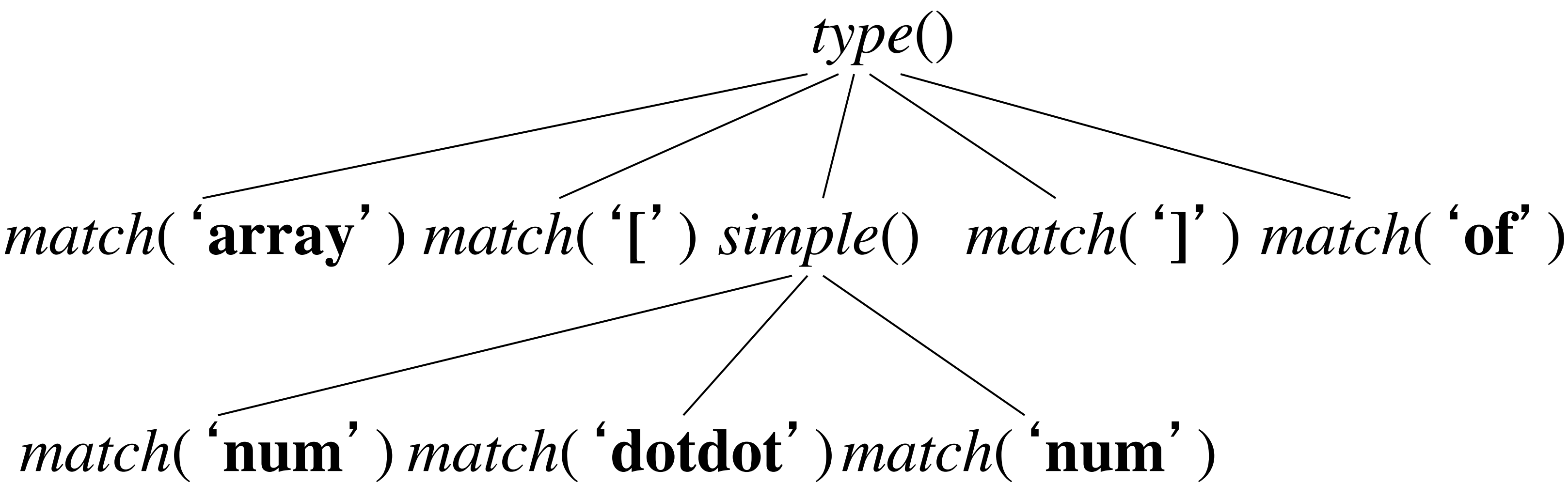
Example Predictive Parser (Execution Step 6)



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

lookahead

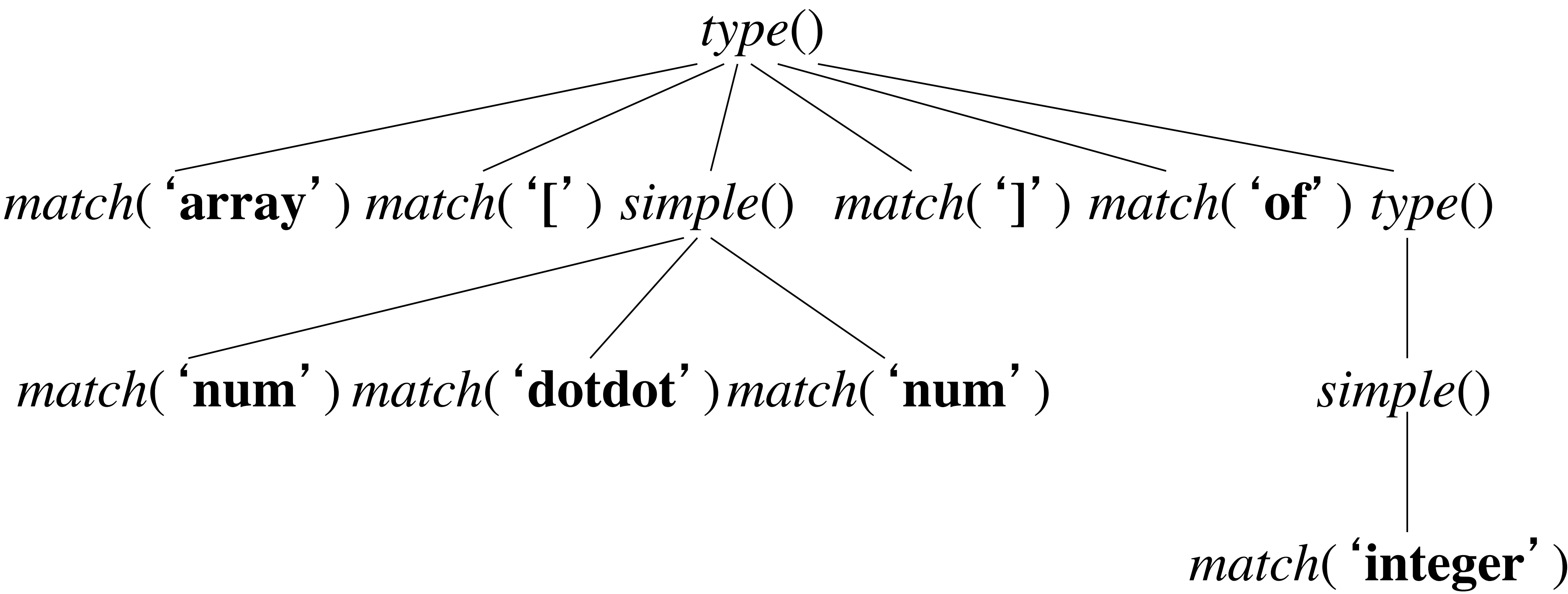
Example Predictive Parser (Execution Step 7)



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

↑ *lookahead*

Example Predictive Parser (Execution Step 8)



<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

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

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*

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

A Translator for Simple Expressions based on Predictive Parsing and Semantic Actions

$expr \rightarrow expr$ **+** $term$ { print("+") }
$expr \rightarrow expr$ **-** $term$ { print("-") }
$expr \rightarrow term$
$term \rightarrow$ **0** { print("0") }
$term \rightarrow$ **1** { print("1") }
… …
$term \rightarrow$ **9** { print("9") }

After left recursion elimination:

$expr \rightarrow term\ rest$
$rest \rightarrow$ **+** $term$ { 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") }

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(); 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

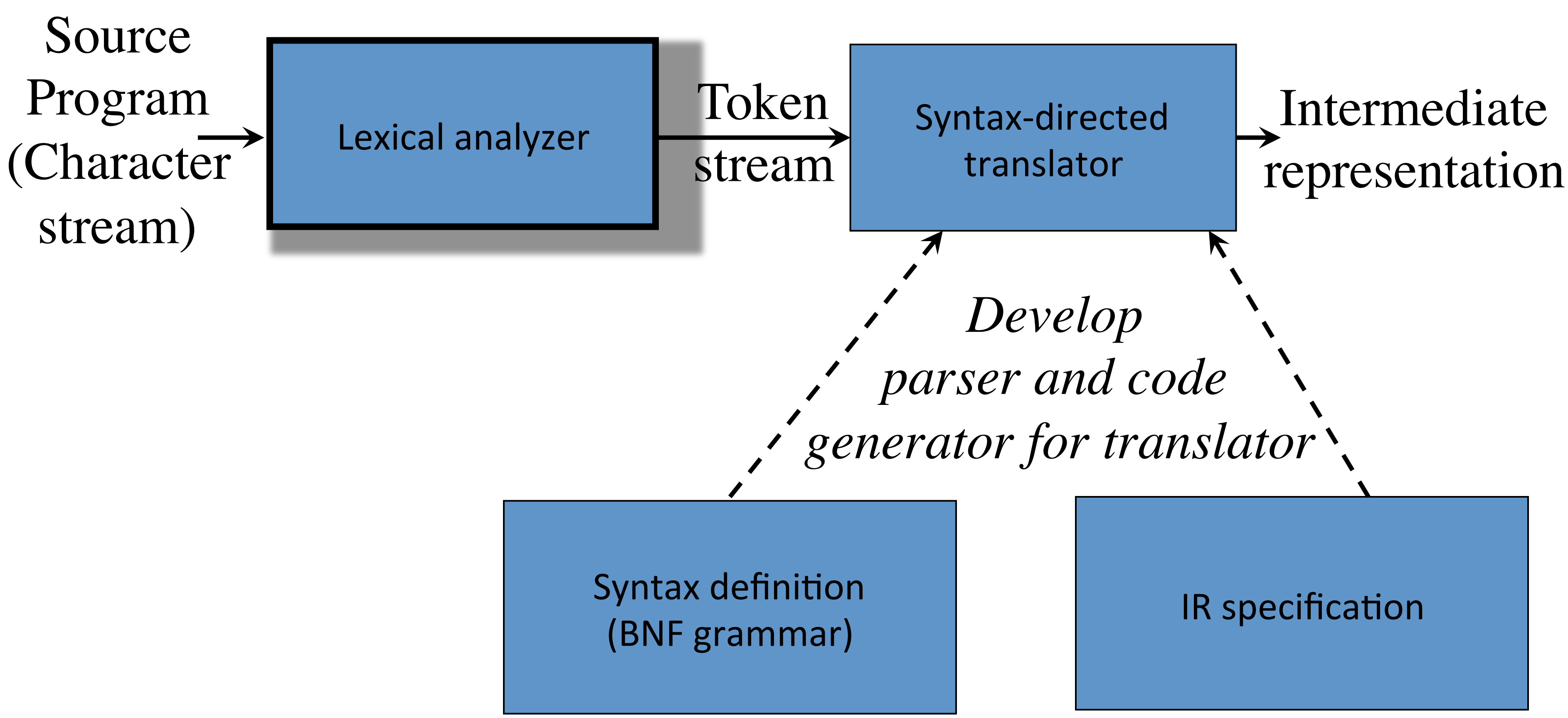
$expr \rightarrow term\ rest$

$rest \rightarrow \mathbf{+}\ term$ { print("+") } $rest$
$rest \rightarrow \mathbf{-}\ term$ { print("-") } $rest$
$rest \rightarrow \varepsilon$

$term \rightarrow \mathbf{0}$ { print("0") }
$term \rightarrow \mathbf{1}$ { print("1") }
…
$term \rightarrow \mathbf{9}$ { 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);
}
```

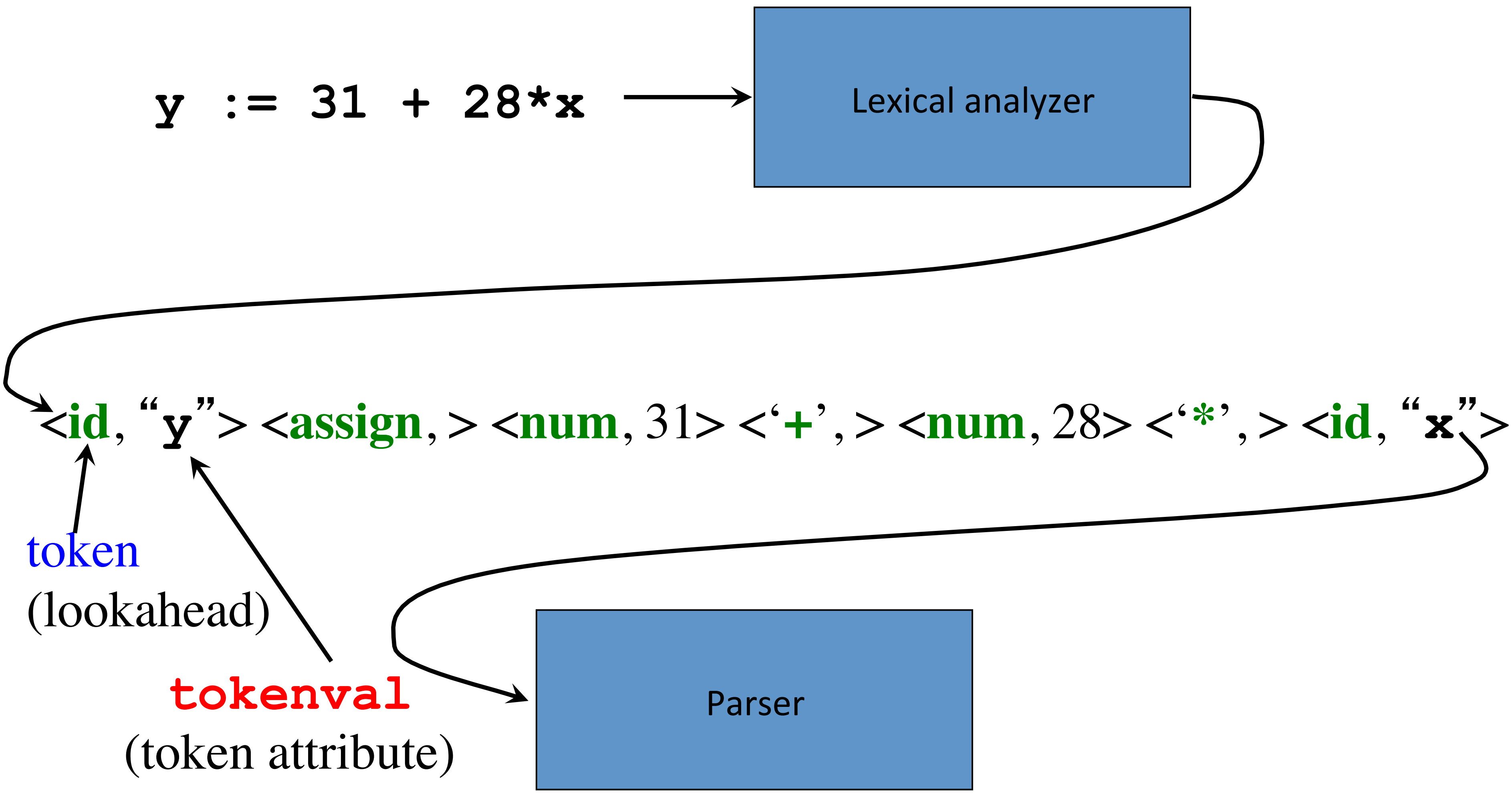
The Structure of the Front-End



Adding a Lexical Analyzer

- Typical tasks of the lexical analyzer:
  - Remove white space and comments
  - Encode constants as tokens
  - Recognize keywords
  - Recognize identifiers and store identifier names in a global **symbol table**

The Lexical Analyzer (“lexer”)



The lookahead of the Parser can be a token, not just a character

Token Attributes

The parser accesses the token via **`lookahead`**, and the token attribute via the global variable **`tokenval`**

$$factor \rightarrow (\ expr\ )$$
$$\quad\quad |\quad \mathbf{num}\ \{\ \mathrm{print}(\mathbf{num}.\mathrm{value})\ \}$$

```
#define NUM 256 /* token returned by lexer */

factor()     /* code of the parser */
{    if (lookahead == '(')
     {    match('('); expr(); match(')');
     }
     else if (lookahead == NUM)
     {    printf(" %d ", tokenval); match(NUM);
     }
     else error();
}
```

Symbol Table

The symbol table is globally accessible (to all phases of the compiler)

Each entry in the symbol table contains a string and a token value:

```
struct entry
{    char *lexptr; /* lexeme (string) for tokenval */
     int token;
};
struct entry symtable[];
```

`insert(s, t)`: returns array index to new entry for string **`s`** token **`t`**

`lookup(s)`: returns array index to entry for string **`s`** or 0

Possible implementations:
- simple C code
- hashtables

Handling identifiers (lexer)

Code executed by the lexer after an identifier has been recognized (stored in **`lexbuf`**):

```
/* lexer.c */
int lexan()
{    …
     tokenval = lookup(lexbuf);
     if (tokenval == 0) /* not found */
          tokenval = insert(lexbuf, ID);
     return symtable[tokenval].token;
}
```

Handling identifiers (parser)

$$\begin{aligned} \mathit{factor} &\rightarrow (\ \mathit{expr}\ ) \\ &\quad \mid \mathbf{id}\ \{\ \mathrm{print}(\mathbf{id}.\mathrm{string})\ \} \end{aligned}$$

```
#define ID 259 /* token returned by lexer */

factor()
{   if (lookahead == '(')
    {   match('('); expr(); match(')');
    }
    else if (lookahead == ID)
    {   printf(" %s ", symtable[tokenval].lexptr);
        match(ID);
    }
    else error();
}
```

provided by the lexer for **ID**

Handling Reserved Keywords (lexer)

Simply initialize the global symbol table with the set of keywords

```
/* global.h */
#define DIV 257 /* token */
#define MOD 258 /* token */
#define ID  259 /* token */

/* init.c */
insert("div", DIV);
insert("mod", MOD);

/* lexer.c */
int lexan()
{    …
     tokenval = lookup(lexbuf);
     if (tokenval == 0) /* not found */
          tokenval = insert(lexbuf, ID);
     return symtable[tokenval].token;
}
```

Handling Reserved Keywords (parser)

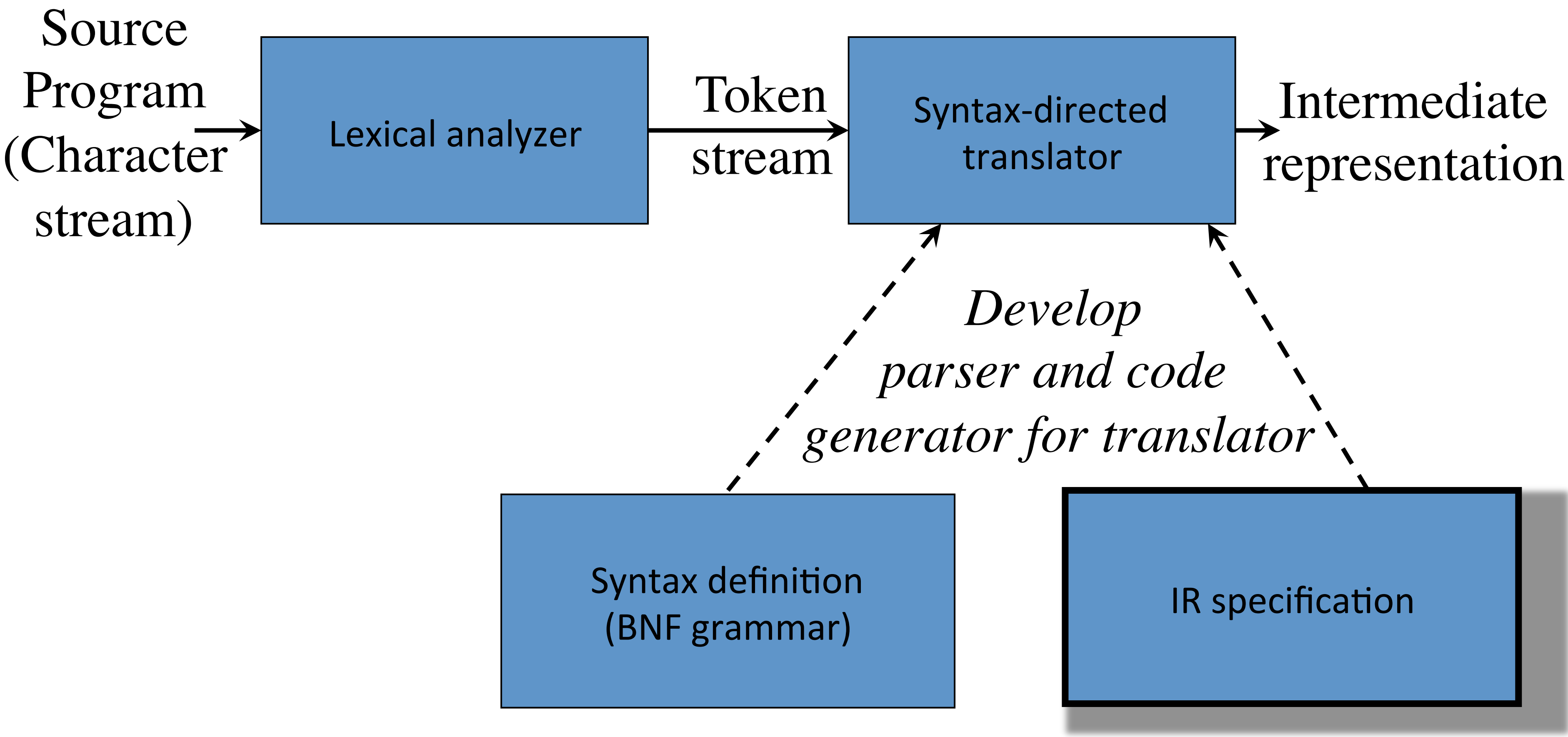
morefactors → **div** *factor* { print(‘DIV’) } *morefactors*
| **mod** *factor* { print(‘MOD’) } *morefactors*
| …

```
/* parser.c */
morefactors()
{   if (lookahead == DIV)
    {   match(DIV); factor(); printf(“DIV”); morefactors();
    }
    else if (lookahead == MOD)
    {   match(MOD); factor(); printf(“MOD”); morefactors();
    }
    else
        …
}
```

Symbol Tables and Scopes

- The same identifier can be declared several times in a program (e.g. in different blocks)
- Each declaration has its own attributes (e.g. type)
- A solution: one Symbol Table per scope
  – Chain of symbol tables for nested blocks
  – Hash table + auxiliary stack (LeBlanc & Cook)
  – Entries have to be created by the parser

The Structure of the Front-End



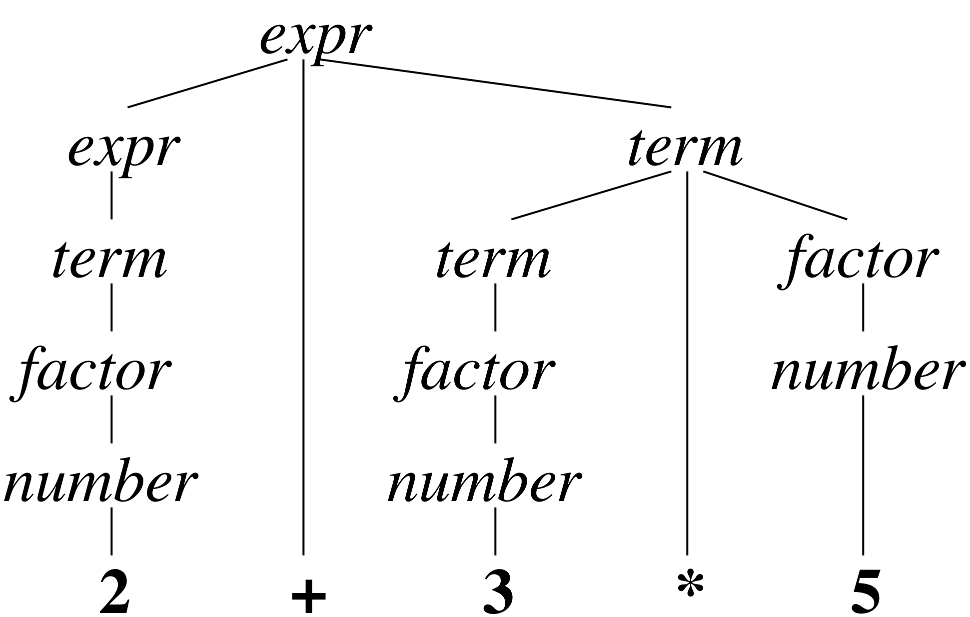
Intermediate Code Generation

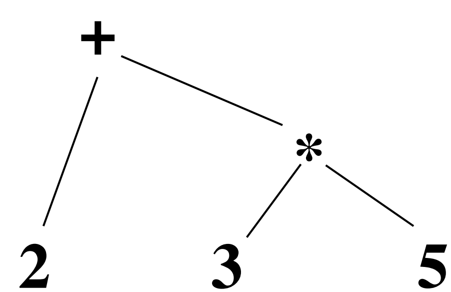
- Two main kinds of intermediate representations:
  - Trees (parse trees, abstract syntax trees)
    - Useful for static semantic analysis ("static checking")
  - Linear representations ("three-address code")
    - Good for machine-independent optimization
- Often compilers produce the linear code during on-the-fly generation of the syntax tree

Abstract Syntax Trees vs. Parse Trees

- **Parse Tree**: tree representation of **concrete syntax** of a program
- **AST**: tree representation of **abstract syntax** of program
  - Brackets are dropped
  - Keywords are dropped, constructs represented by nodes

$$
\begin{aligned}
\mathit{expr} &\rightarrow \mathit{expr} \;\mathbf{+}\; \mathit{term} \mid \mathit{term} \\
\mathit{term} &\rightarrow \mathit{term} \;\mathbf{*}\; \mathit{factor} \mid \mathit{factor} \\
\mathit{factor} &\rightarrow \mathit{number} \mid \mathbf{(}\; \mathit{expr} \;\mathbf{)}
\end{aligned}
$$





Translation Scheme for generating the AST during parsing: Expressions

- Each node of the Parse Tree “points” to a node of the AST
- Each operator is a node of the AST, with “semantically meaningful components” as children
- Semantic actions either build a new node of the AST (with suitable parameters), or return the node of the only subexpression

$$
\begin{array}{rcll}
\mathit{expr} & \rightarrow & \mathit{rel} \texttt{ = } \mathit{expr}_1 & \{\ \mathit{expr.n} = \textbf{new}\ \mathit{Assign}('=', \mathit{rel.n}, \mathit{expr}_1.n);\ \} \\
& | & \mathit{rel} & \{\ \mathit{expr.n} = \mathit{rel.n};\ \} \\
\\
\mathit{rel} & \rightarrow & \mathit{rel}_1 \texttt{ < } \mathit{add} & \{\ \mathit{rel.n} = \textbf{new}\ \mathit{Rel}('<', \mathit{rel}_1.n, \mathit{add.n});\ \} \\
& | & \mathit{rel}_1 \texttt{ <= } \mathit{add} & \{\ \mathit{rel.n} = \textbf{new}\ \mathit{Rel}('\leq', \mathit{rel}_1.n, \mathit{add.n});\ \} \\
& | & \mathit{add} & \{\ \mathit{rel.n} = \mathit{add.n};\ \} \\
\\
\mathit{add} & \rightarrow & \mathit{add}_1 \texttt{ + } \mathit{term} & \{\ \mathit{add.n} = \textbf{new}\ \mathit{Op}('+', \mathit{add}_1.n, \mathit{term.n});\ \} \\
& | & \mathit{term} & \{\ \mathit{add.n} = \mathit{term.n};\ \} \\
\\
\mathit{term} & \rightarrow & \mathit{term}_1 \texttt{ * } \mathit{factor} & \{\ \mathit{term.n} = \textbf{new}\ \mathit{Op}('*', \mathit{term}_1.n, \mathit{factor.n});\ \} \\
& | & \mathit{factor} & \{\ \mathit{term.n} = \mathit{factor.n};\ \} \\
\\
\mathit{factor} & \rightarrow & \texttt{(}\ \mathit{expr}\ \texttt{)} & \{\ \mathit{factor.n} = \mathit{expr.n};\ \} \\
& | & \textbf{num} & \{\ \mathit{factor.n} = \textbf{new}\ \mathit{Num}(\textbf{num}.\mathit{value});\ \}
\end{array}
$$

Useless nonterminals are not represented

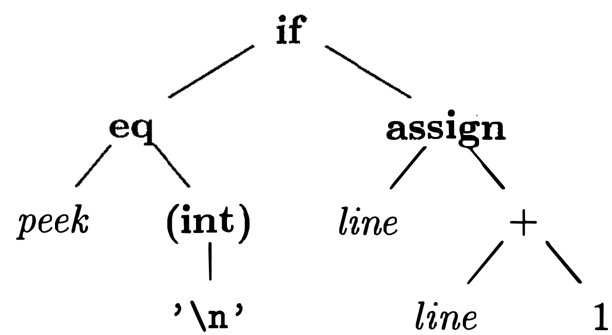
Drop brackets

Translation Scheme for generating the Abstract Syntax Tree: Statements

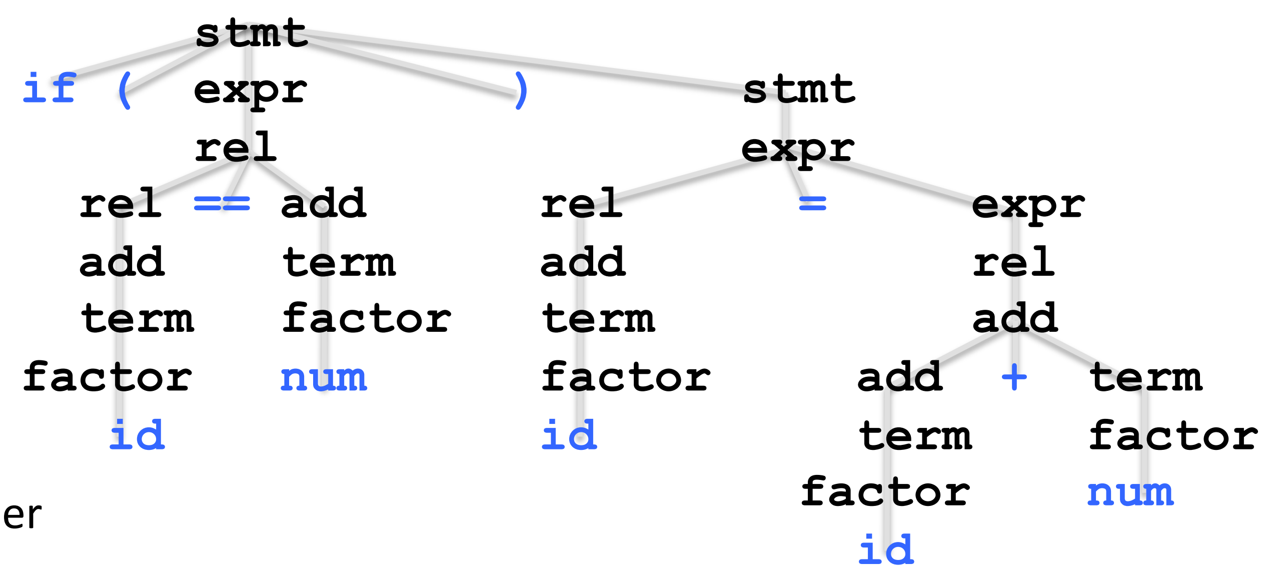
- Statements as operators
- The concrete syntax (keywords) is dropped

$$
\begin{array}{rcll}
program & \rightarrow & block & \{ \text{ return } block.n; \ \} \\
\\
block & \rightarrow & '\{' \ stmts \ '\}' & \{ \ block.n = stmts.n; \ \} \\
\\
stmts & \rightarrow & stmts_1 \ \ stmt & \{ \ stmts.n = \mathbf{new} \ Seq(stmts_1.n, stmt.n); \ \} \\
 & | & \epsilon & \{ \ stmts.n = \mathbf{null}; \ \} \\
\\
stmt & \rightarrow & expr \ ; & \{ \ stmt.n = \mathbf{new} \ Eval(expr.n); \ \} \\
 & | & \mathbf{if} \ (\ expr \ ) \ stmt_1 & \\
 & & & \{ \ stmt.n = \mathbf{new} \ If(expr.n, stmt_1.n); \ \} \\
 & | & \mathbf{while} \ (\ expr \ ) \ stmt_1 & \\
 & & & \{ \ stmt.n = \mathbf{new} \ While(expr.n, stmt_1.n); \ \} \\
 & | & \mathbf{do} \ stmt_1 \ \mathbf{while} \ (\ expr \ ); & \\
 & & & \{ \ stmt.n = \mathbf{new} \ Do(stmt_1.n, expr.n); \ \} \\
 & | & block & \{ \ stmt.n = block.n; \ \}
\end{array}
$$

An example: from a statement to the abstract syntax tree



Generation of Abstract Syntax Tree



Parser

```
<if> <(> <id, "peek"> <eq> <const, '\n'> <)>
       <id, "line"> <assign> <id, "line"> <+> <num, 1> <;>
```

Scanner

```
if ( peek == '\n' ) line = line + 1;
```

Static Checking

- **Syntactic properties** (not captured by the context-free grammar of the language) are checked by analyzing the parse tree or the abstract syntax tree
- **Context-dependent syntactic** properties:
  1. Every variable is declared before used
  2. Each identifier is declared at most once per scope
  3. Left operands of assignments are L-values
  4. Break statements must have enclosing loop or switch
- **Semantic analysis** is applied by the compiler to discover the meaning of a program by analyzing its parse tree or abstract syntax tree. Useful to prevent runtime errors.
- **Static semantic** checks performed at compile time:
  - Type checking: each operator is applied to arguments of the right type
  - Handling of coercion and overloading

Semantic Analysis

- Dynamic semantic checks are performed at run time, and the compiler produces code that performs these checks
  - Array subscript values are within bounds
  - Arithmetic errors, e.g. division by zero
  - Pointers are not dereferenced unless pointing to valid object
  - A variable is used but hasn't been initialized
  - When a check fails at run time, an exception is raised

Generation of Three Address Code

- Linear Intermediate Representation generated by structural induction executing a function on the nodes of the tree
- Sequence of instructions of the form

```
x = y op z
```

- Arrays handled with instructions

```
x[y] = z
x = y[z]
```

- Sequence control handled with jump instructions:

```
ifFalse x goto L
ifTrue x goto L
goto L
```

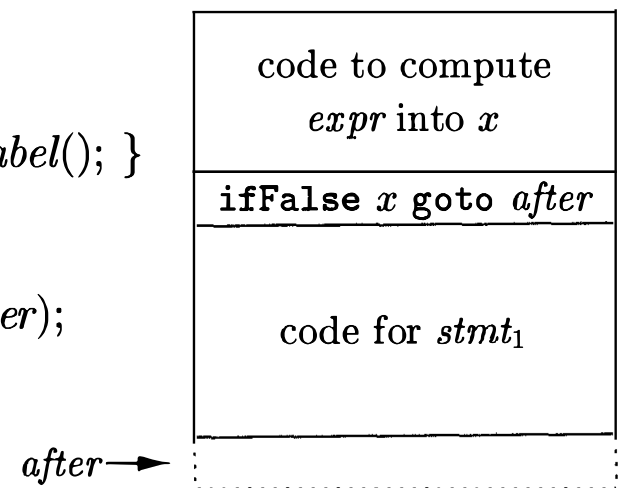
- Statements may have any number of labels, e.g.

```
L1:L2: x = y
```

Translation of Statements

- Jumps are used to implement the control flow
- Example: **`if`** ***`expr`*** **`then`** ***`stmt`***$_1$ is translated to

```
class If extends Stmt {
        Expr E; Stmt S;
        public If(Expr x, Stmt y) { E = x; S = y; after = newlabel(); }
        public void gen() {
                Expr n = E.rvalue();
                emit( "ifFalse " + n.toString() + " goto " + after);
                S.gen();
                emit(after + ":");
        }
}
```



Translation of expressions

- One operation at a time, using temporaries

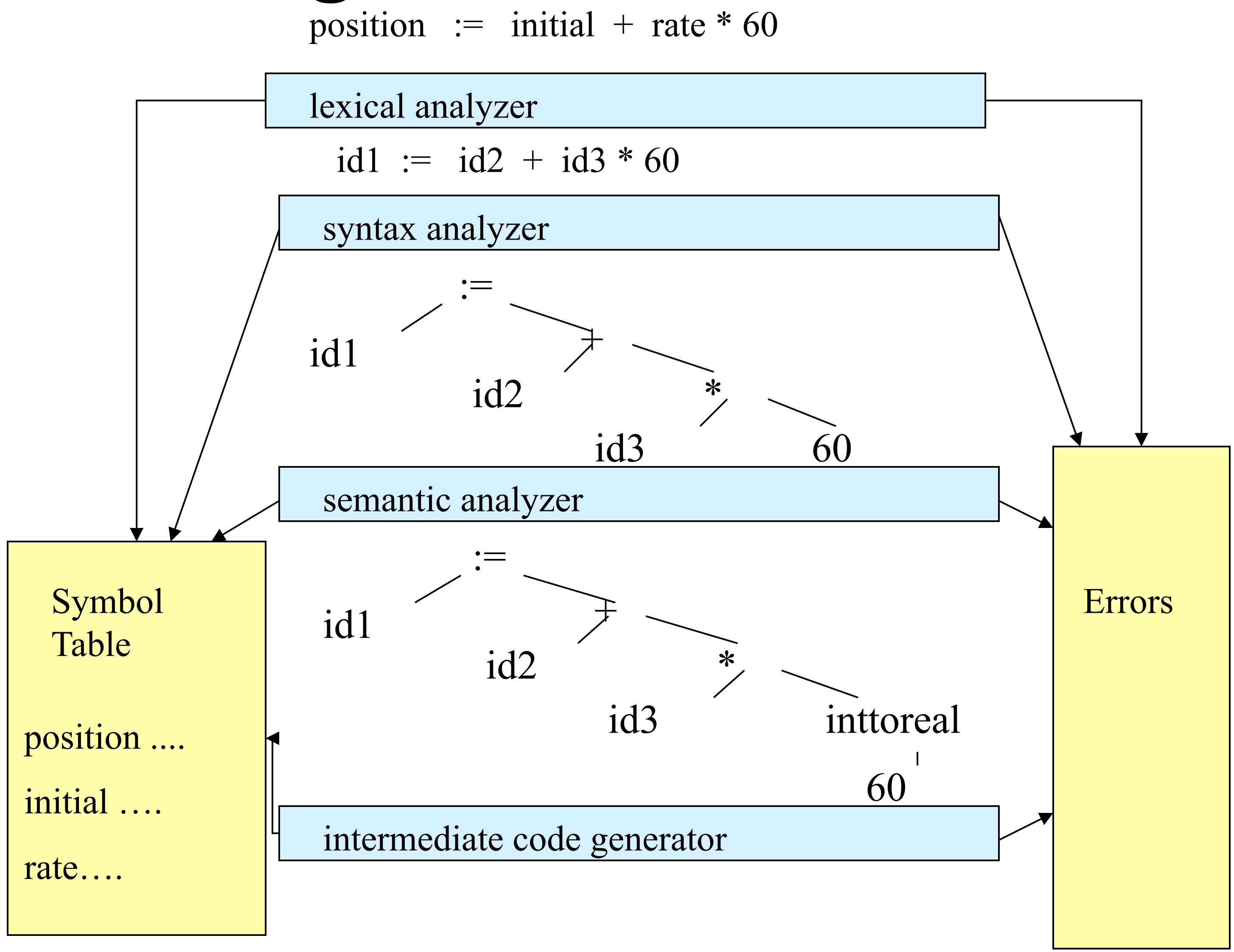
```
i-j+k  →  t1 = i - j
          t2 = t1 + k
```

```
2*a[i]  →  t1 = a[i]
           t2 = 2 * t1
```

- L-values in assignments cannot be translated into temporaries

```
a[i]=2*a[j-k]  →  t3 = j - k
                  t2 = a [ t3 ]
                  t1 = 2 * t2
                  a [ i ] = t1
```

Reviewing the Entire Process



Reviewing the Entire Process