Principles of Programming Languages

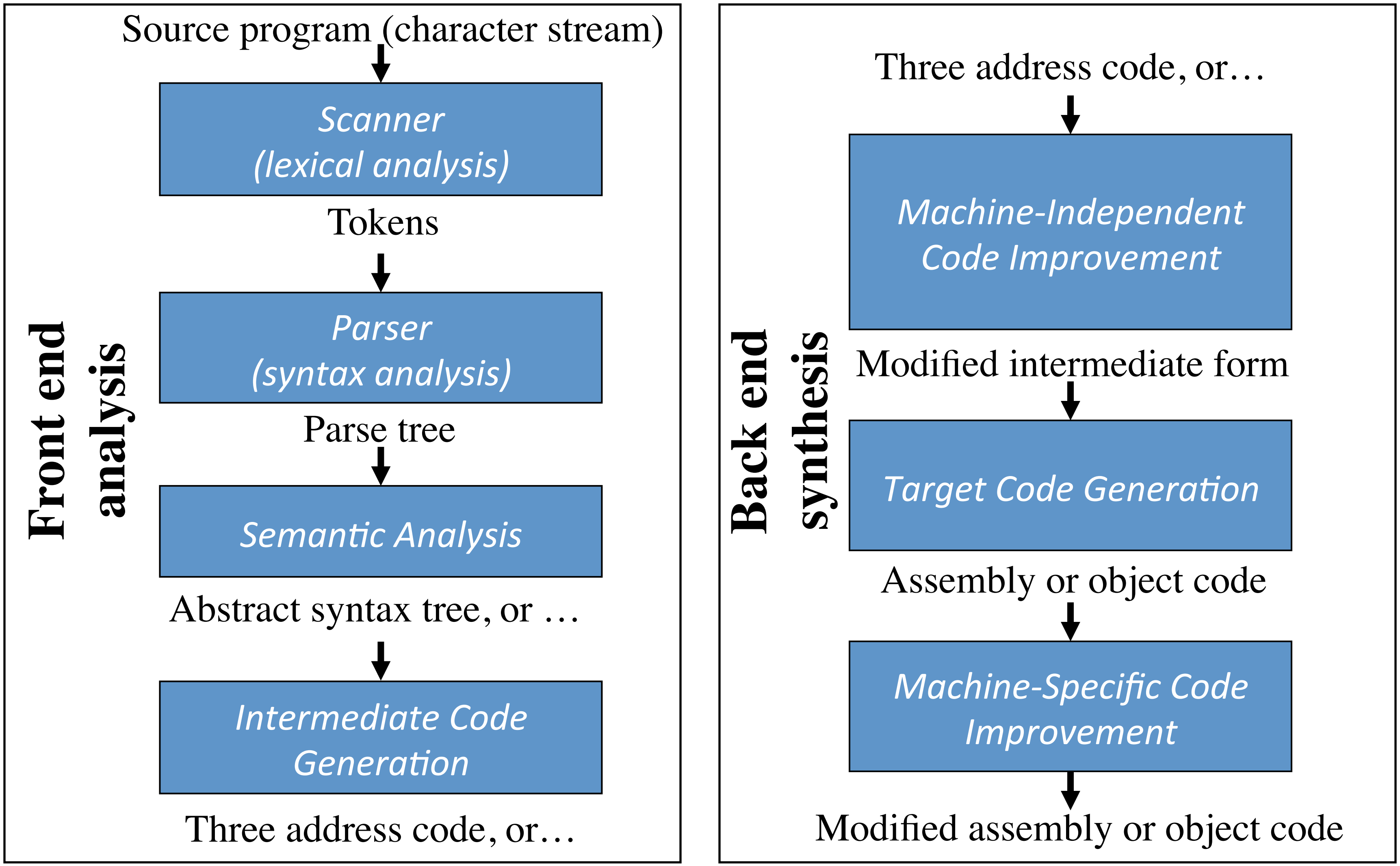
http://www.di.unipi.it/~andrea/Didattica/PLP-15/

Prof. Andrea Corradini
Department of Computer Science, Pisa

Lesson 4

- Overview of a Simple Compiler Front-end
  - Lexical analysis
  - Intermediate code generation
  - Static checking

Compiler Front- and Back-end



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

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

After left recursion elimination:

$expr \rightarrow term \; rest$
$rest \rightarrow \mathbf{+} \; term$ { 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") }

Code of the translator

$expr \rightarrow term\ rest$

$rest \rightarrow \mathbf{+}\ term\ \{\ \mathrm{print}(\text{“+”})\ \}\ rest$

$rest \rightarrow \mathbf{-}\ term\ \{\ \mathrm{print}(\text{“-”})\ \}\ rest$

$rest \rightarrow \varepsilon$

$term \rightarrow \mathbf{0}\ \{\ \mathrm{print}(\text{“0”})\ \}$

$term \rightarrow \mathbf{1}\ \{\ \mathrm{print}(\text{“1”})\ \}$

…

$term \rightarrow \mathbf{9}\ \{\ \mathrm{print}(\text{“9”})\ \}$

```
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

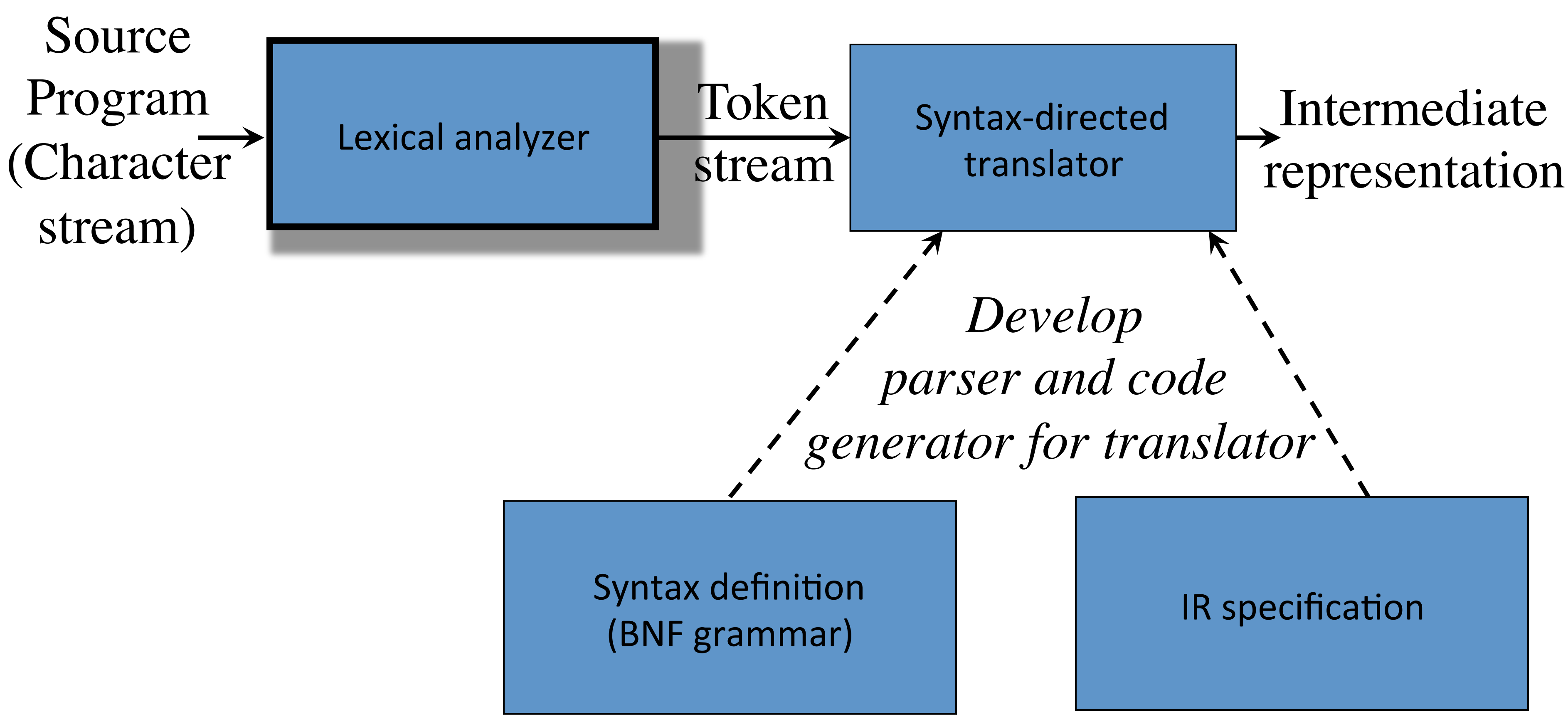
$$expr \rightarrow term\ rest$$

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

$$term \rightarrow \mathbf{0}\ \{\ \text{print}(\text{"0"})\ \}$$
$$term \rightarrow \mathbf{1}\ \{\ \text{print}(\text{"1"})\ \}$$
$$\ldots$$
$$term \rightarrow \mathbf{9}\ \{\ \text{print}(\text{"9"})\ \}$$

```
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);
}
```

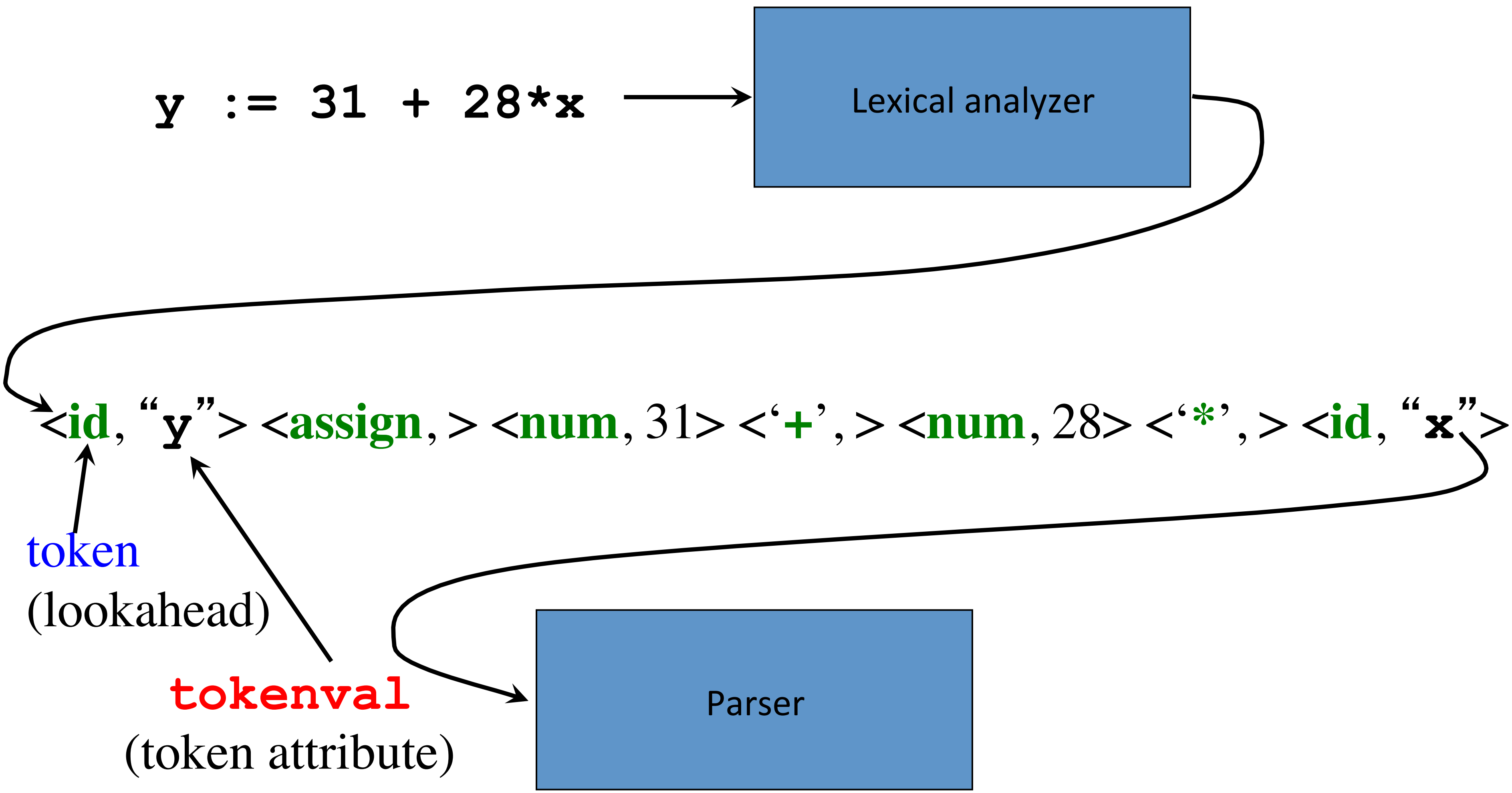
The Structure of the Front-End



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 → (*expr*)
        | **num** { print(**num**.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)

$$\mathit{factor} \rightarrow (\ \mathit{expr}\ )$$
$$\quad\quad |\ \mathbf{id}\ \{\ \mathrm{print}(\mathbf{id}.\mathrm{string})\ \}$$

```
#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)

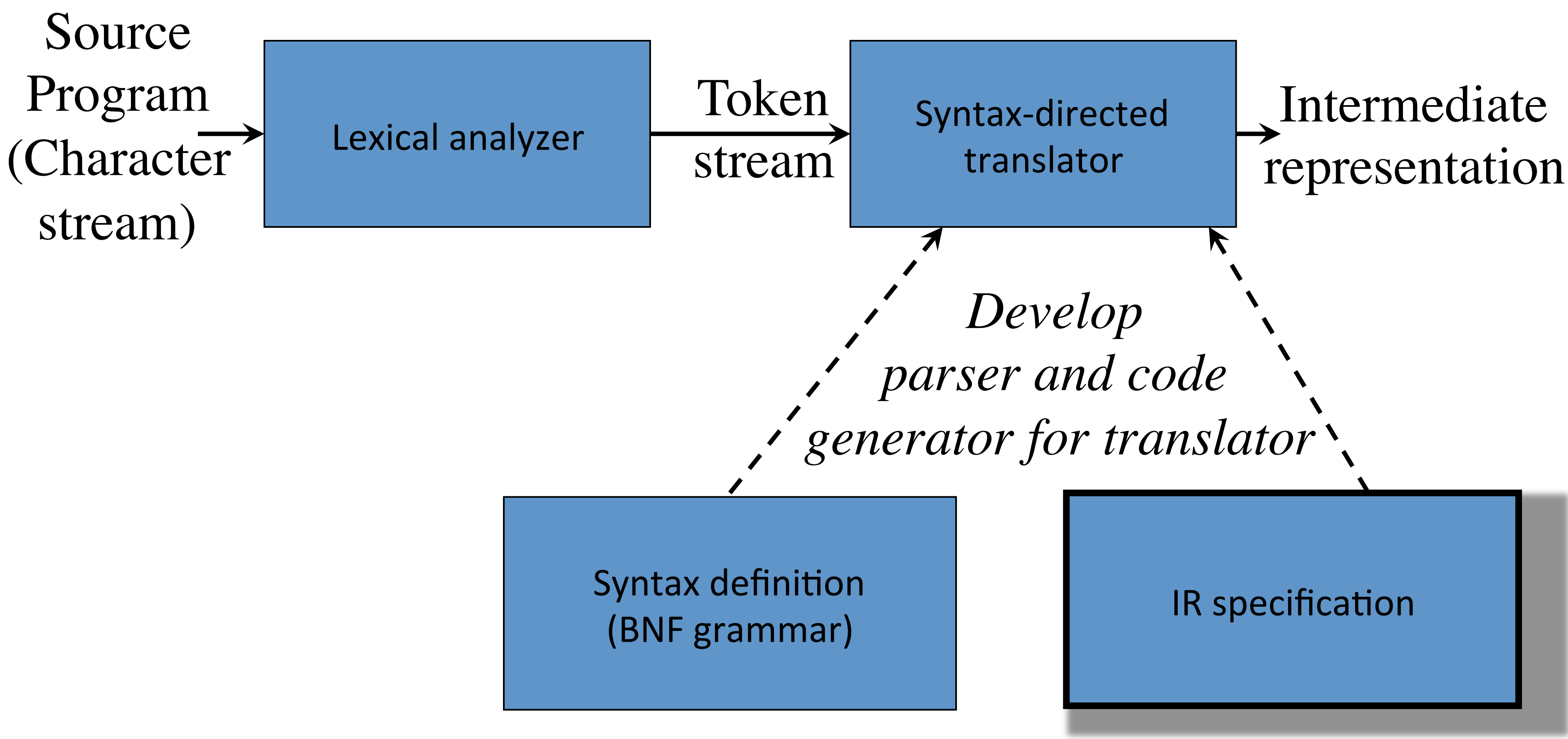
morefactors → **div** *factor* { print('DIV') } *morefactors*
| **mod** *factor* { print('MOD') } *morefactors*
| …

```
/* 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
  - Entries have to be created by the parser

The Structure of the Front-End

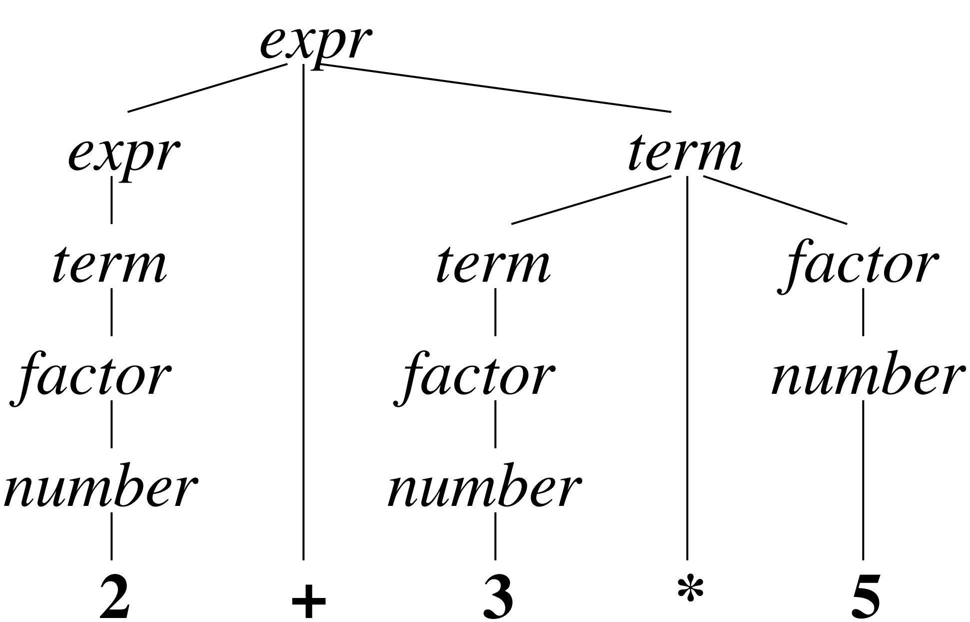


Intermediate Code Generation

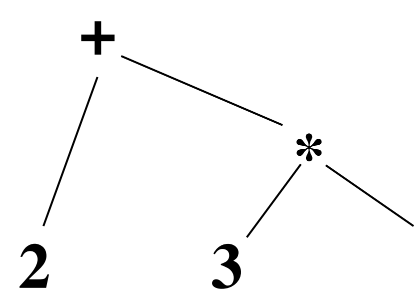
- Two main kinds of intermediate representations:
  - Trees (parse trees, abstract syntax trees)
    - Useful for static semantic analysis (“static checking”)
  - Linear representations (“three-address code”)
    - Good for machine-independent optimization
- Often compilers produce the linear code during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



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



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

$$
\begin{array}{rcll}
expr & \rightarrow & rel \texttt{ = } expr_1 & \{\ expr.n = \mathbf{new}\ Assign('=', rel.n, expr_1.n);\ \} \\
 & | & rel & \{\ expr.n = rel.n;\ \} \\
 & & & \\
rel & \rightarrow & rel_1 \texttt{ < } add & \{\ rel.n = \mathbf{new}\ Rel('<', rel_1.n, add.n);\ \} \\
 & | & rel_1 \texttt{ <= } add & \{\ rel.n = \mathbf{new}\ Rel('\leq', rel_1.n, add.n);\ \} \\
 & | & add & \{\ rel.n = add.n;\ \} \\
 & & & \\
add & \rightarrow & add_1 \texttt{ + } term & \{\ add.n = \mathbf{new}\ Op('+', add_1.n, term.n);\ \} \\
 & | & term & \{\ add.n = term.n;\ \} \\
 & & & \\
term & \rightarrow & term_1 * factor & \{\ term.n = \mathbf{new}\ Op('*', term_1.n, factor.n);\ \} \\
 & | & factor & \{\ term.n = factor.n;\ \} \\
 & & & \\
factor & \rightarrow & (\ expr\ ) & \{\ factor.n = expr.n;\ \} \\
 & | & \mathbf{num} & \{\ factor.n = \mathbf{new}\ Num(\mathbf{num}.value);\ \}
\end{array}
$$

Useless nonterminals are not represented

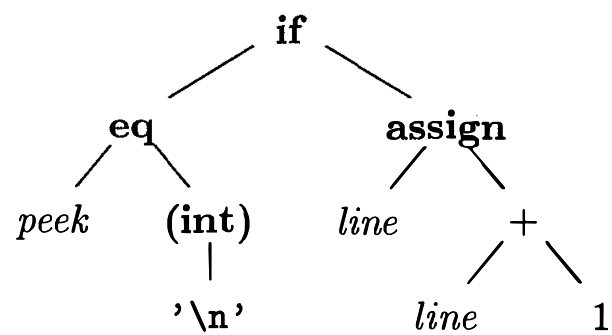
Drop brackets

Translation Scheme for generating the Abstract Syntax Tree: Statements

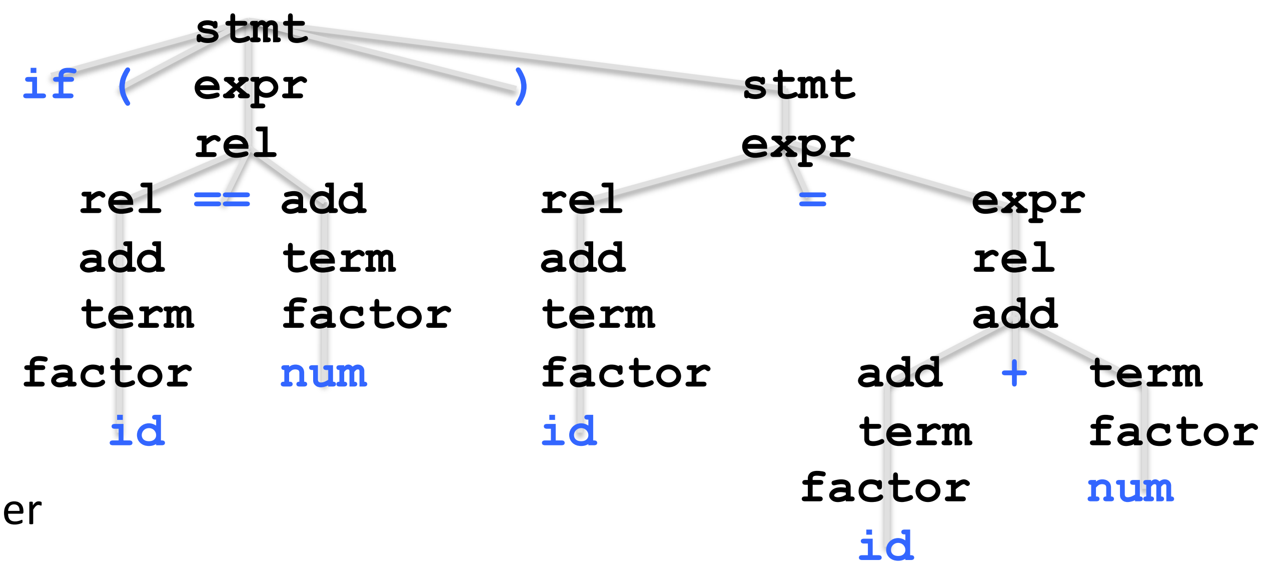
- Statements as operators
- The concrete syntax (keywords) is dropped

$$
\begin{array}{rcll}
\mathit{program} & \rightarrow & \mathit{block} & \{ \text{ return } \mathit{block.n}; \} \\
\\
\mathit{block} & \rightarrow & '\{' \ \mathit{stmts} \ '\}' & \{ \mathit{block.n} = \mathit{stmts.n}; \} \\
\\
\mathit{stmts} & \rightarrow & \mathit{stmts}_1 \ \mathit{stmt} & \{ \mathit{stmts.n} = \textbf{new } \mathit{Seq}(\mathit{stmts}_1.n, \mathit{stmt.n}); \} \\
& | & \epsilon & \{ \mathit{stmts.n} = \textbf{null}; \} \\
\\
\mathit{stmt} & \rightarrow & \mathit{expr} \ ; & \{ \mathit{stmt.n} = \textbf{new } \mathit{Eval}(\mathit{expr.n}); \} \\
& | & \textbf{if} \ (\ \mathit{expr} \ ) \ \mathit{stmt}_1 & \\
& & & \{ \mathit{stmt.n} = \textbf{new } \mathit{If}(\mathit{expr.n}, \mathit{stmt}_1.n); \} \\
& | & \textbf{while} \ (\ \mathit{expr} \ ) \ \mathit{stmt}_1 & \\
& & & \{ \mathit{stmt.n} = \textbf{new } \mathit{While}(\mathit{expr.n}, \mathit{stmt}_1.n); \} \\
& | & \textbf{do} \ \mathit{stmt}_1 \ \textbf{while} \ (\ \mathit{expr} \ ) ; & \\
& & & \{ \mathit{stmt.n} = \textbf{new } \mathit{Do}(\mathit{stmt}_1.n, \mathit{expr.n}); \} \\
& | & \mathit{block} & \{ \mathit{stmt.n} = \mathit{block.n}; \}
\end{array}
$$

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

Exercise

- For each of the numbered items in the last slide, discuss how the property can be checked either with a translation scheme or with suitable attributes of the parse tree

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
```

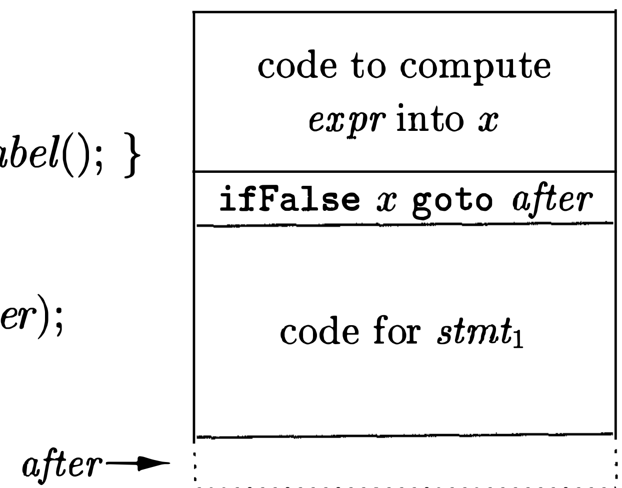
- Statements may have any number of labels, e.g.

```
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

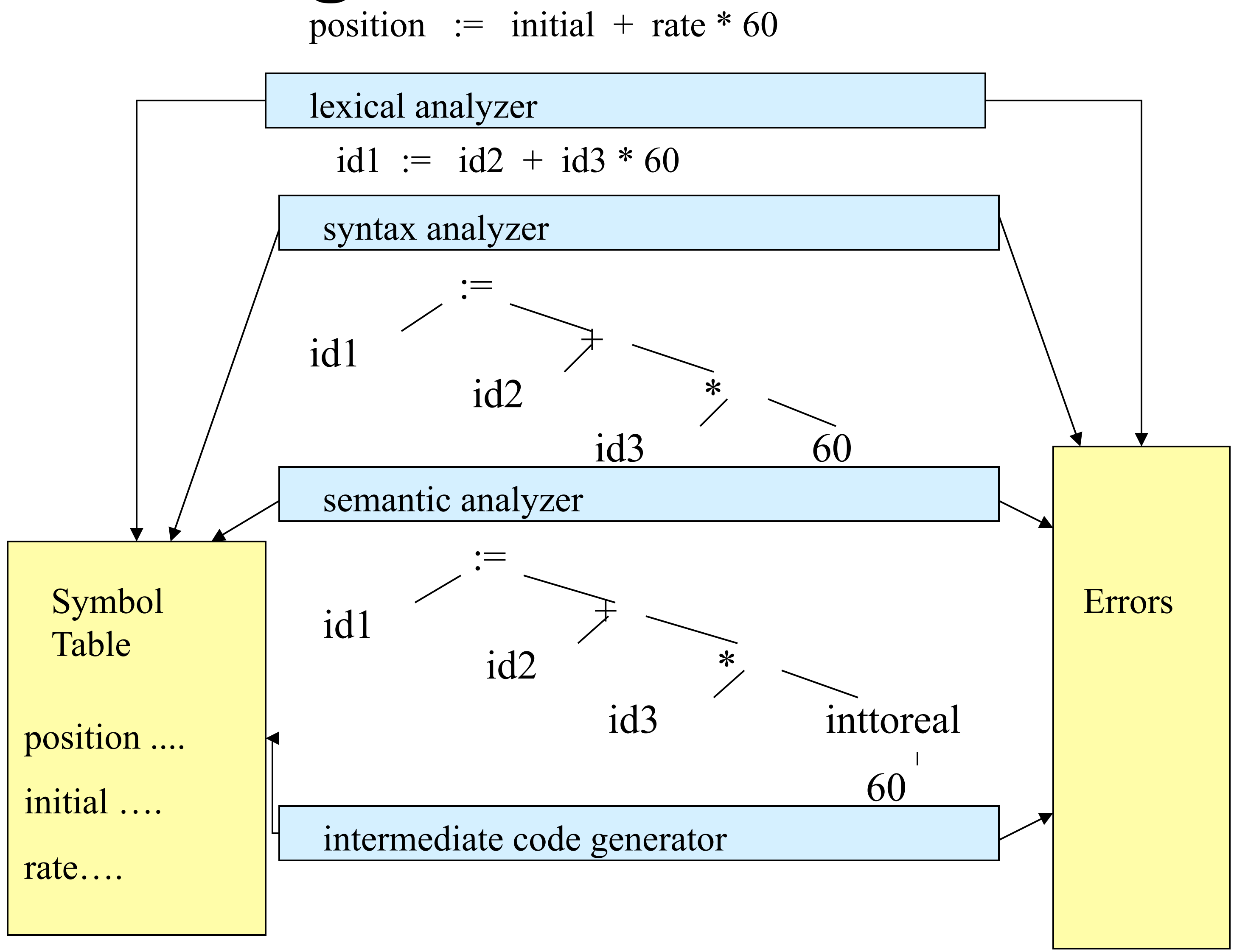
```
i-j+k  →  t1 = i - j
          t2 = t1 + k

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