Principles of Programming Languages

http://www.di.unipi.it/~andrea/Didattica/PLP-14/
Prof. Andrea Corradini
Department of Computer Science, Pisa

Lesson 3

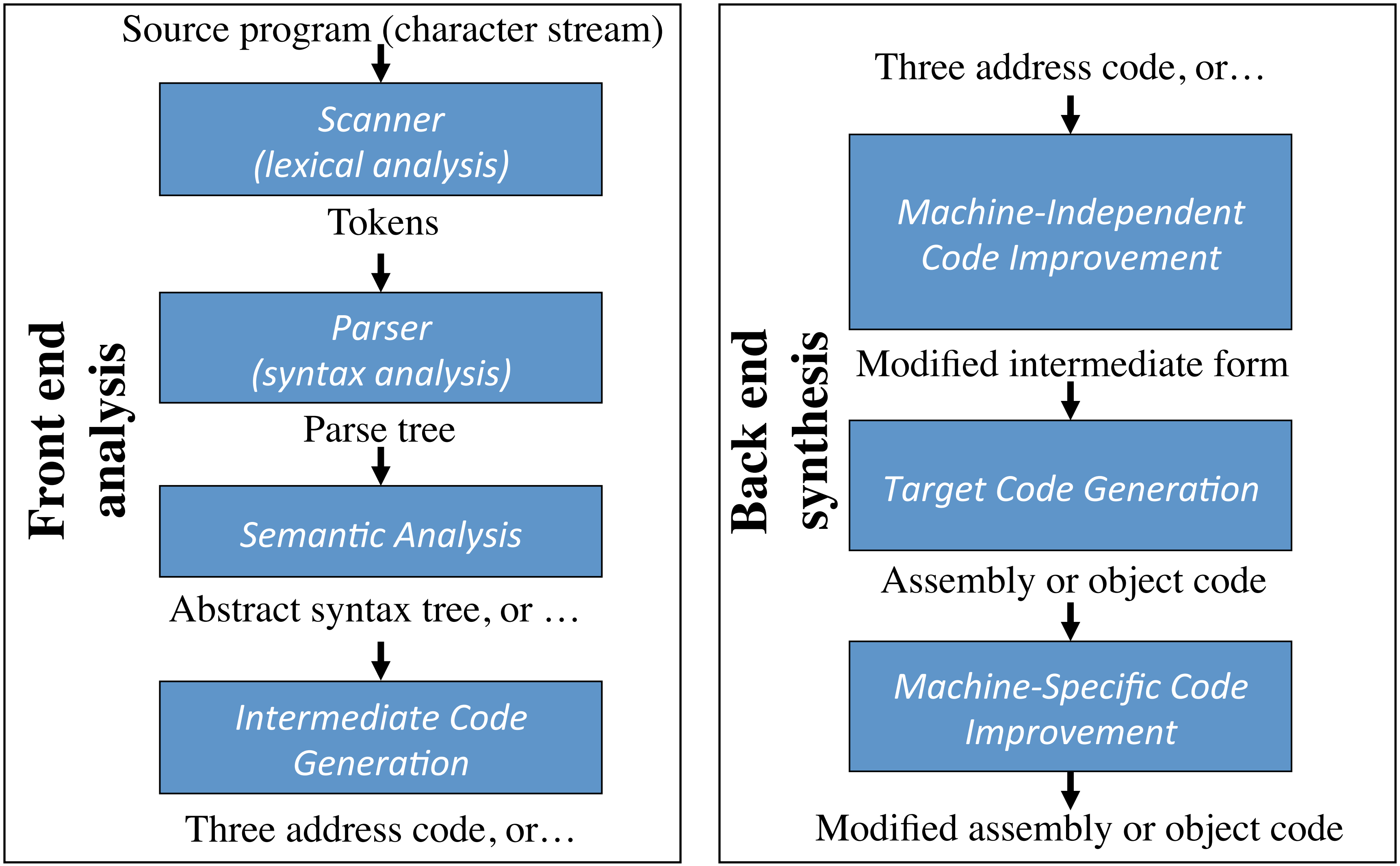
- Overview of a Simple Compiler Front-end
  - Lexical analysis
  - Intermediate code generation
  - Static checking

Admins

- Office Hours:
  - Wednesday, 9 - 11

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

Compiler Front- and Back-end



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

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") }

Optimized code of the translator

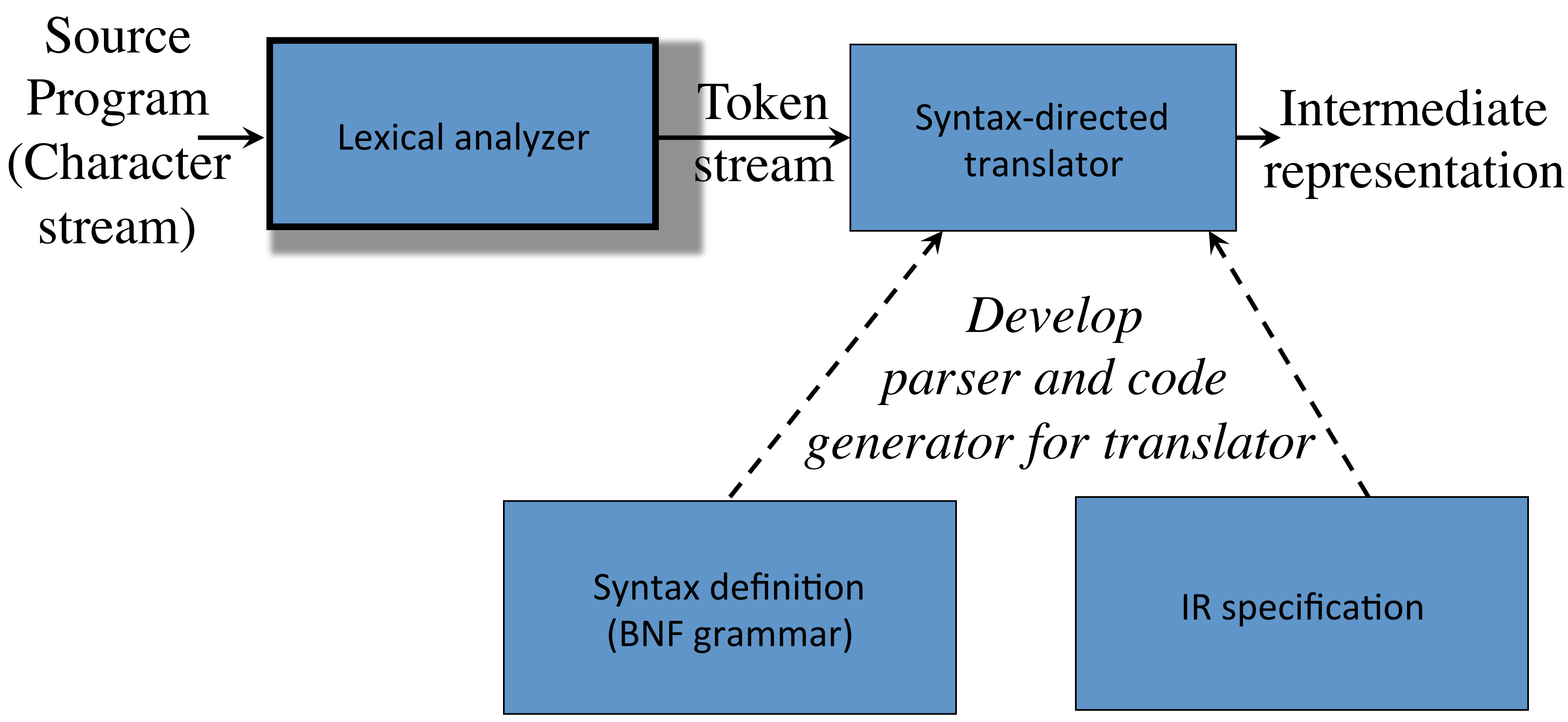
$expr \rightarrow term\ rest$

$rest \rightarrow$ **+** $term$ { 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();
    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);
}
```

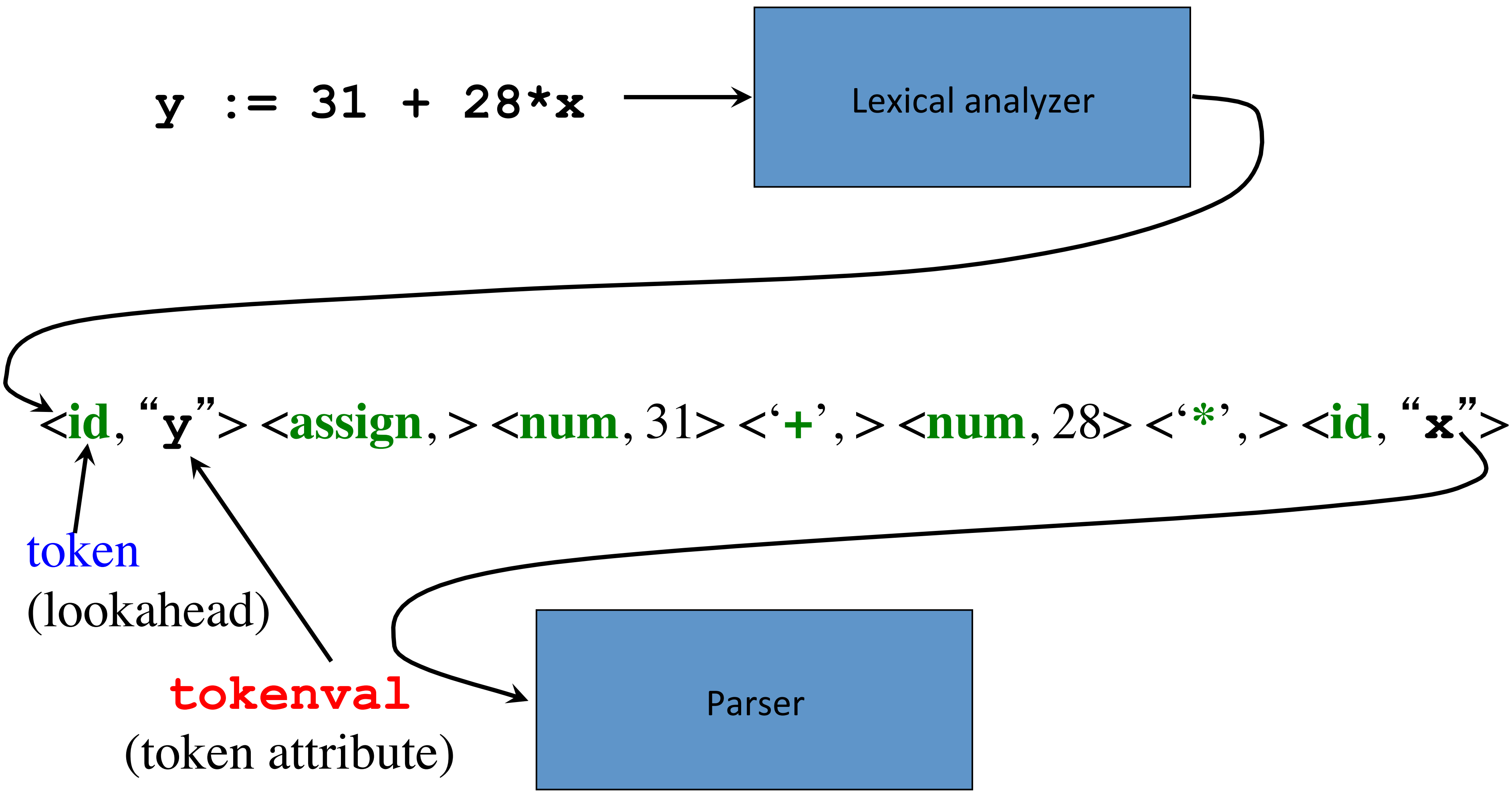
The Structure of the Front-End



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`**

$$factor \rightarrow (\ expr\ )$$
$$\quad\quad \mid \mathbf{num}\ \{\ \text{print}(\mathbf{num}.\text{value})\ \}$$

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

factor()
{   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)

$$factor \rightarrow (\ expr\ )$$
$$\mid \mathbf{id}\ \{\ \text{print}(\mathbf{id}.\text{string})\ \}$$

```
#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)

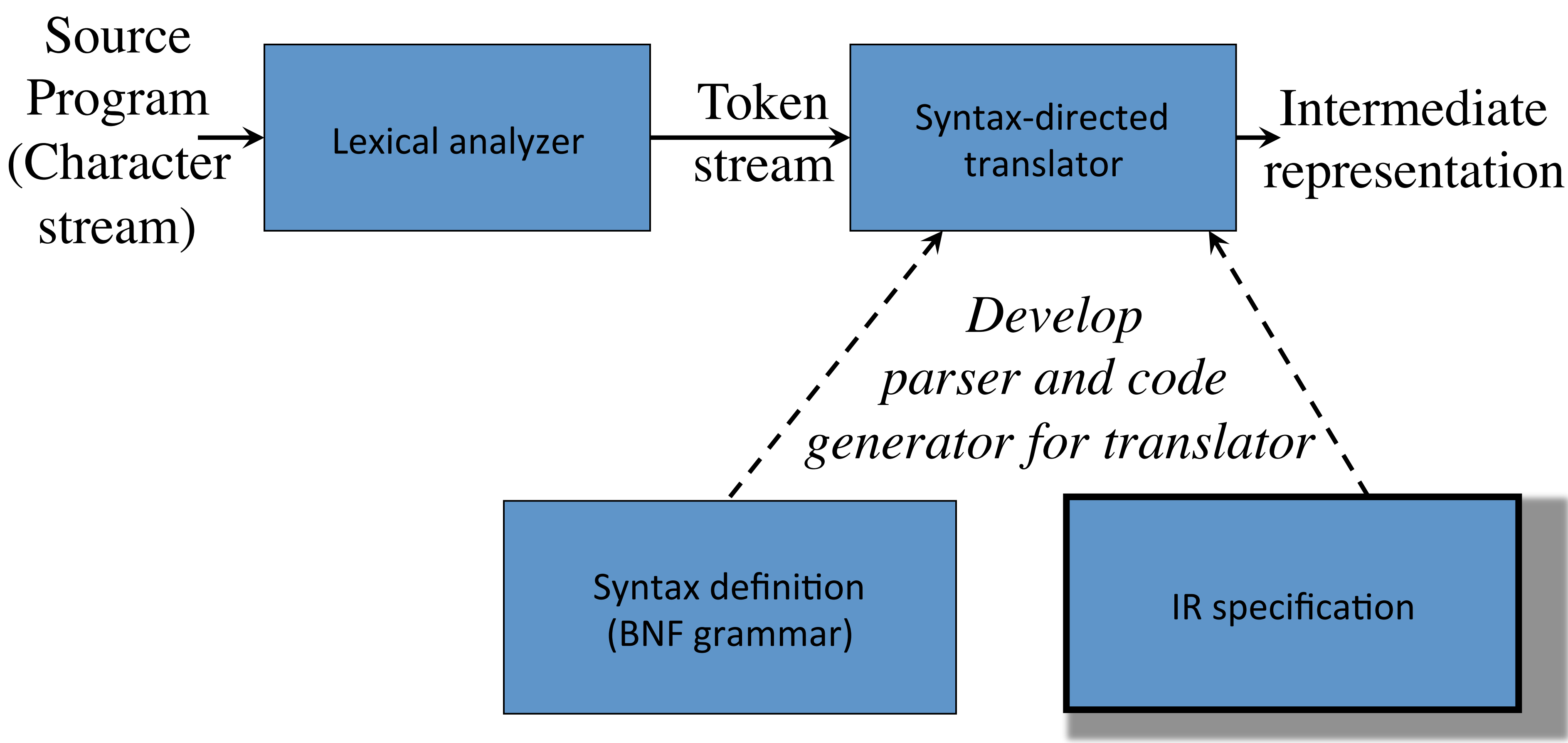
morefactors → **div** *factor* { print('DIV') } *morefactors*
| **mod** *factor* { print('MOD') } *morefactors*
| …

```
/* 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

Translation Scheme for generating the Abstract Syntax Tree: Expressions

- Each operator is a node, with "semantically meaningful components" as children
- For each production the semantic action either builds a new node (with suitable parameters), or returns the node of the only subexpression

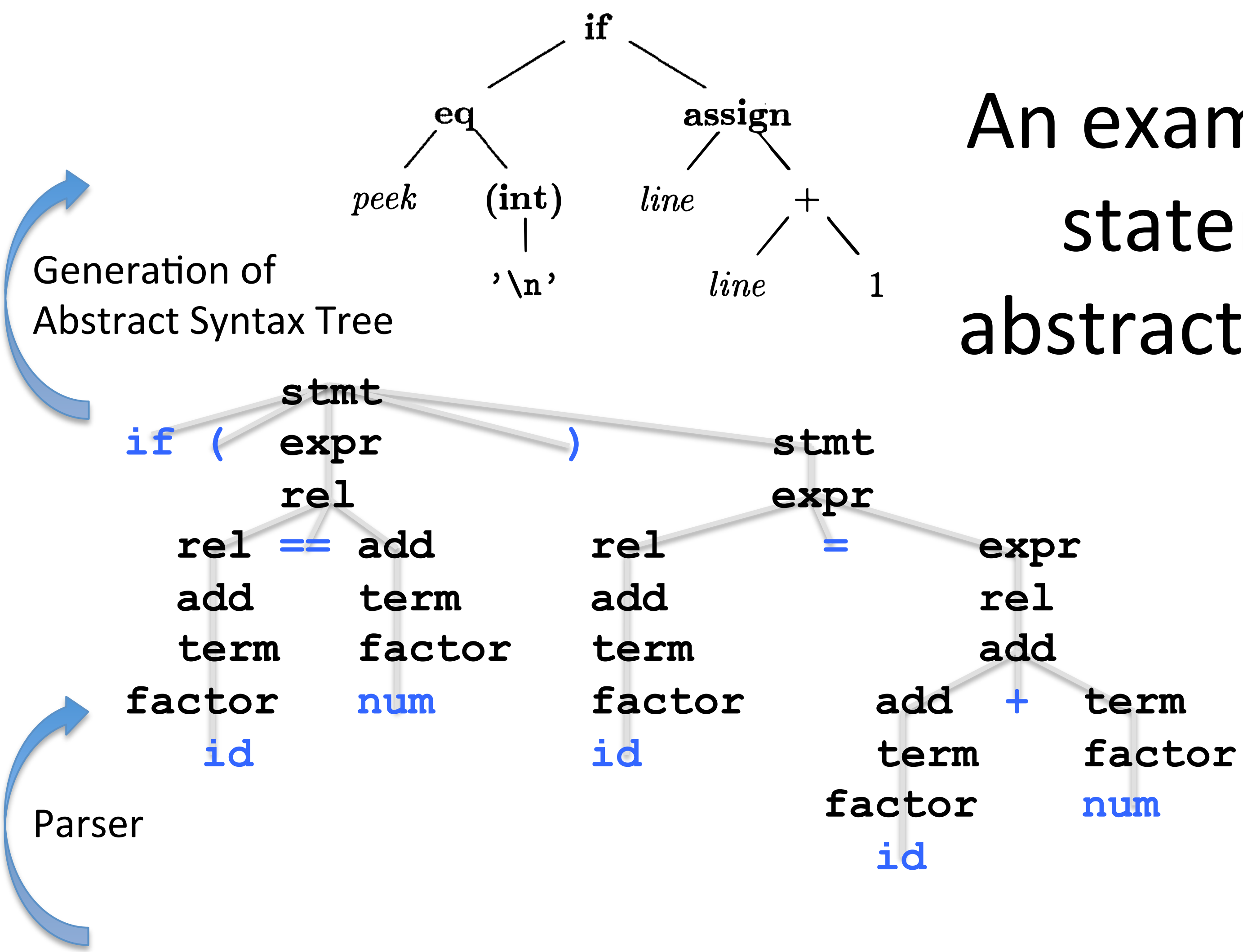
$$
\begin{array}{rcll}
\mathit{expr} & \rightarrow & \mathit{rel} \ \texttt{=} \ \mathit{expr}_1 & \{ \ \mathit{expr.n} = \textbf{new} \ \mathit{Assign}('=', \mathit{rel.n}, \mathit{expr}_1.n); \ \} \\
 & | & \mathit{rel} & \{ \ \mathit{expr.n} = \mathit{rel.n}; \ \} \\
 \\
\mathit{rel} & \rightarrow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

Translation Scheme for generating the Abstract Syntax Tree: Statements

- Statements as operators: note that the concrete syntax 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




```
<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:
  5. 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
```

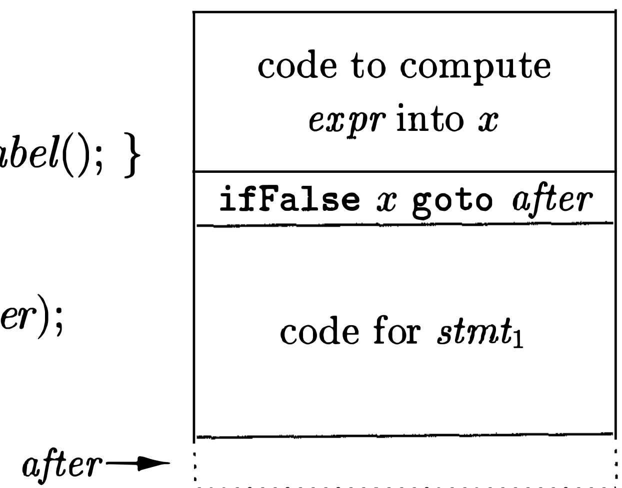
- Statement may have any number of labels, e.g.

```
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 assignements cannot be translated into temporaries

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

Translation of expressions: the pseudocode for L-value and R-value

```
Expr lvalue(x : Expr) {
    if ( x is an Id node ) return x;
    else if ( x is an Access(y, z) node and y is an Id node ) {
        return new Access(y, rvalue(z));
    }
    else error;
}
```

```
Expr rvalue(x : Expr) {
    if ( x is an Id or a Constant node ) return x;
    else if ( x is an Op(op, y, z) or a Rel(op, y, z) node ) {
        t = new temporary;
        emit string for t = rvalue(y) op rvalue(z);
        return a new node for t;
    }
    else if ( x is an Access(y, z) node ) {
        t = new temporary;
        call lvalue(x), which returns Access(y, z');
        emit string for t = Access(y, z');
        return a new node for t;
    }
    else if ( x is an Assign(y, z) node ) {
        z' = rvalue(z);
        emit string for lvalue(y) = z';
        return z';
    }
}
```

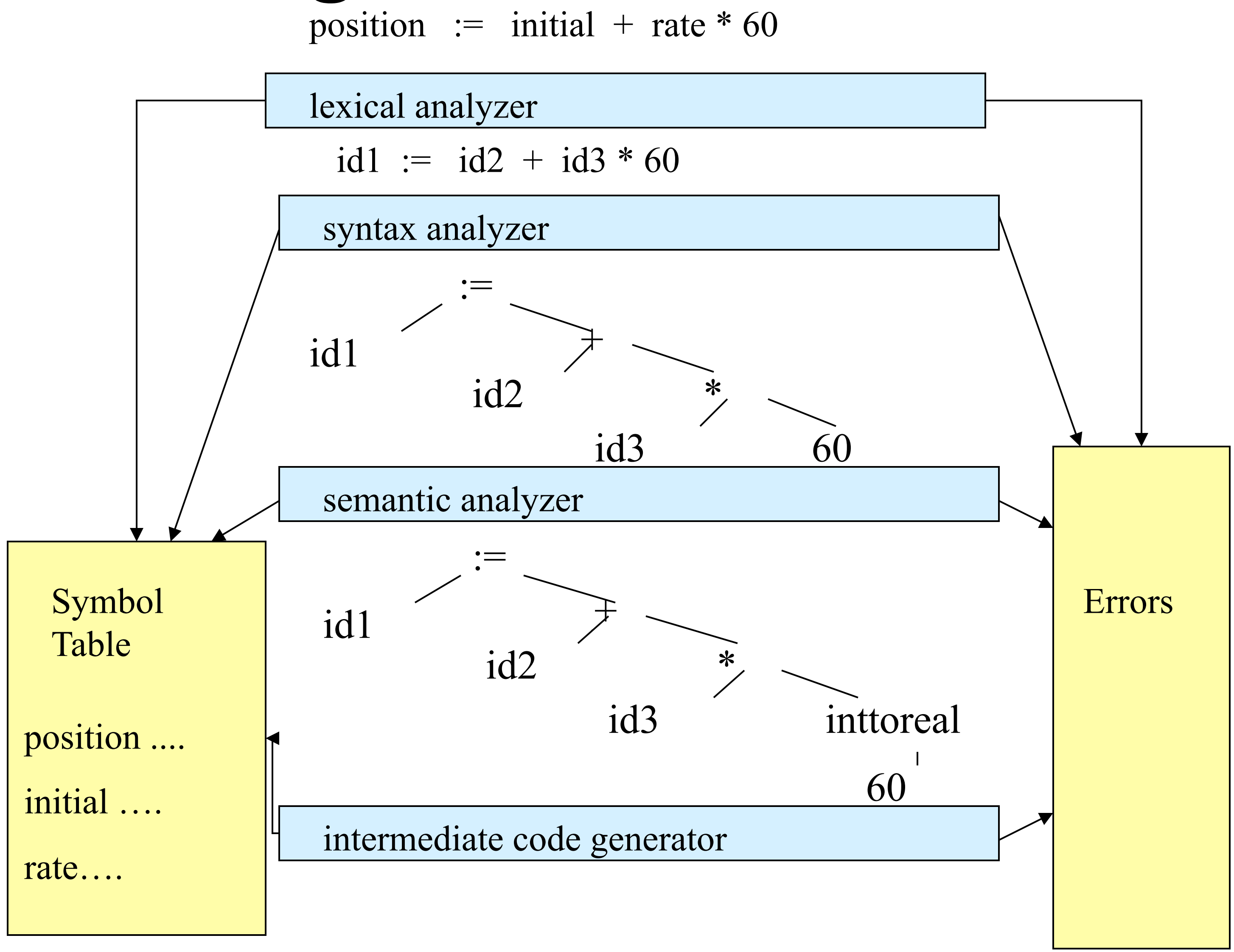
The Phases of a Compiler

Phase	Output	Sample
Programmer (source code producer)	Source string	**`A=B+C;`**
Scanner (performs *lexical analysis*)	Token string	**`'A', '=', 'B', '+', 'C', ';'`** And *symbol table* with names

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)	Annotated parse tree or abstract syntax tree	
Intermediate code generator	Three-address code, quads, or RTL (Register Transfer Language)	**`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`**

Reviewing the Entire Process



Reviewing the Entire Process