Honors Compilers

Amir Pnueli

Mondays, Wednesdays 2:00-3:15 PM

Copies of presentations and Lecture Notes will be available at
http://www.cs.nyu.edu/courses/spring07/G22.3130-001/index.htm

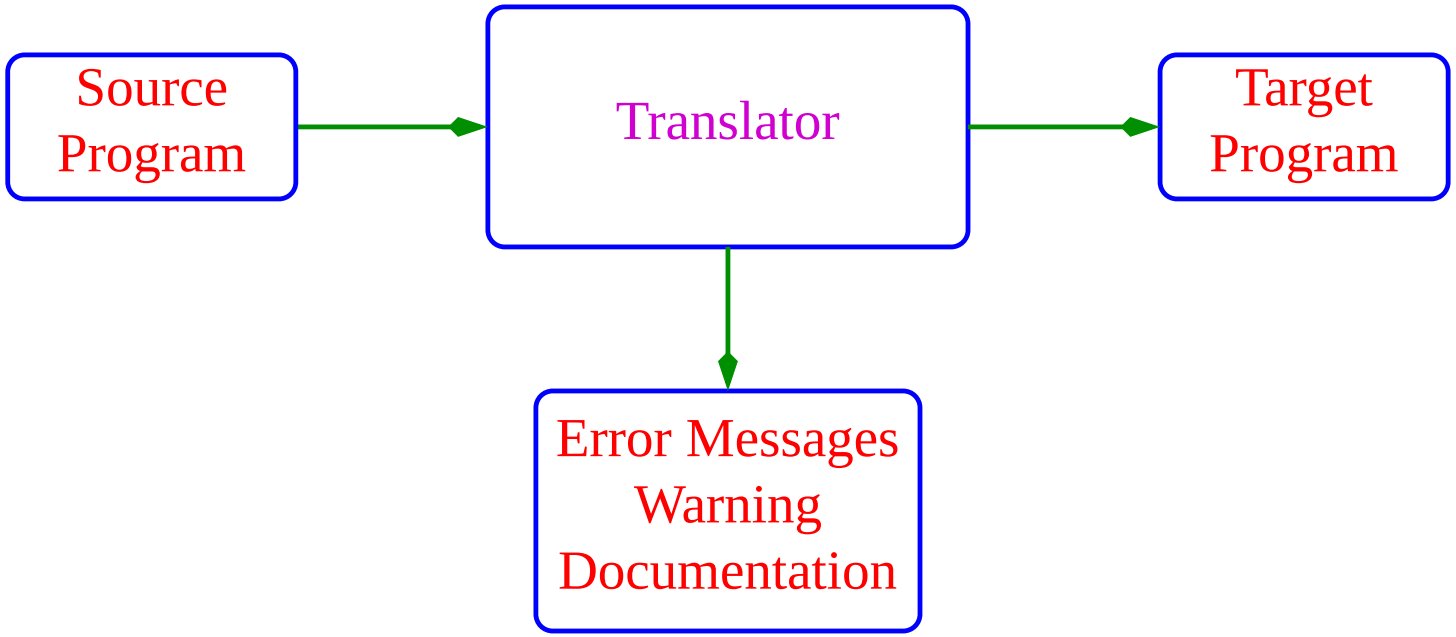
Please register at the course's class list:
http://www.cs.nyu.edu/mailman/listinfo/g22_3130_001_sp07

Required Textbook: Compilers, principles, Techniques & tools
by Aho, Lam, Sethi, and Ullman, Addison Wesley.

Additional recommended readings will be listed in the course's web page.

What is a Compiler?

Compiler: A translator from a source to a target program.



Why Study Compiler Construction?

We do not expect many of you to become compiler builders.

However,

- Many applications (structure-sensitive editors, pretty printers, etc.) use components of a compiler, e.g. analysis and translation.
- The study of compilers clarifies many deep issues in programming languages and their execution, e.g. recursion, multi-threading. It may help you design your own mini-language.
- Understanding a compiler and its optimization mechanisms enable us to write more efficient programs.

For example, the optimization —

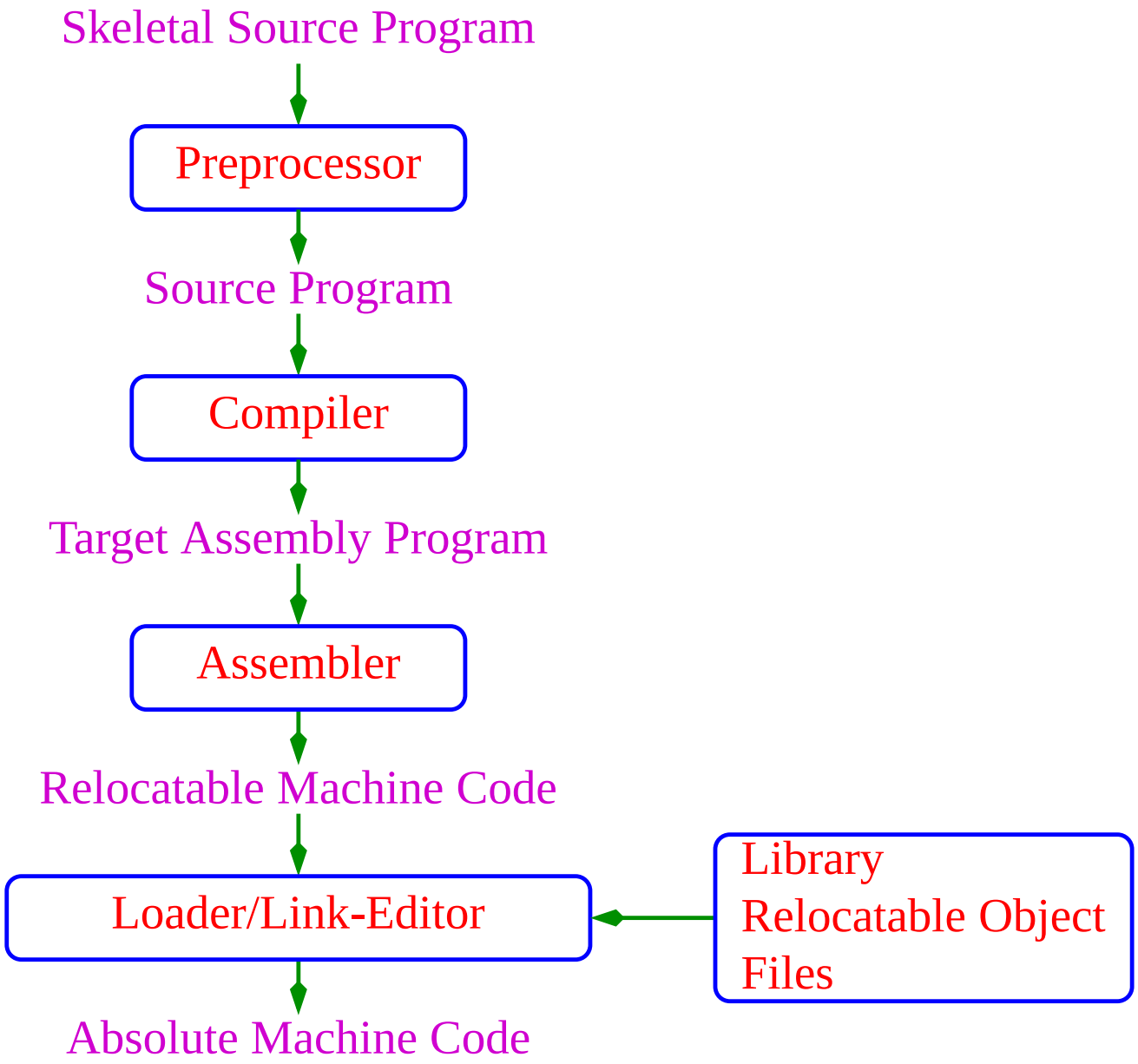
for $i = 1$ **to** 100000 **do**
  $A := B/C$

$\Longrightarrow$

$A := B/C;$
for $i = 1$ **to** 100000 **do**
  skip

Context of a Compiler

A compiler is often applied as a stage within a sequence of transformations:



Analysis of the Source Program

Analysis can be partitioned into three phases:

- Linear (Lexical) Analysis. Stream of characters is read left-to-right and partitioned into tokens.
- Hierarchical (Syntax) Analysis. Tokens are grouped hierarchically into nested collections.

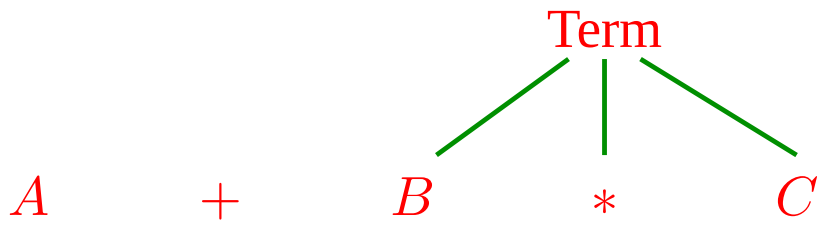
$$A \quad + \quad B \quad * \quad C$$

- Semantic Analysis. Checking global consistency. Often does not comply with hierarchical structure. Type checking is an instance of such analysis.

Analysis of the Source Program

Analysis can be partitioned into three phases:

- Linear (Lexical) Analysis. Stream of characters is read left-to-right and partitioned into tokens.
- Hierarchical (Syntax) Analysis. Tokens are grouped hierarchically into nested collections.

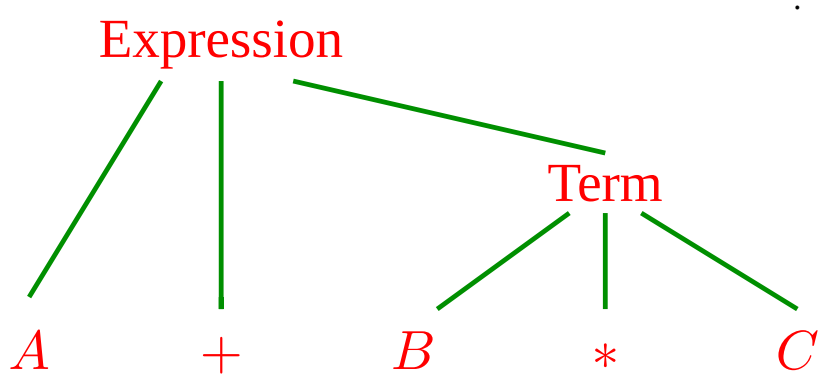


- Semantic Analysis. Checking global consistency. Often does not comply with hierarchical structure. Type checking is an instance of such analysis.

Analysis of the Source Program

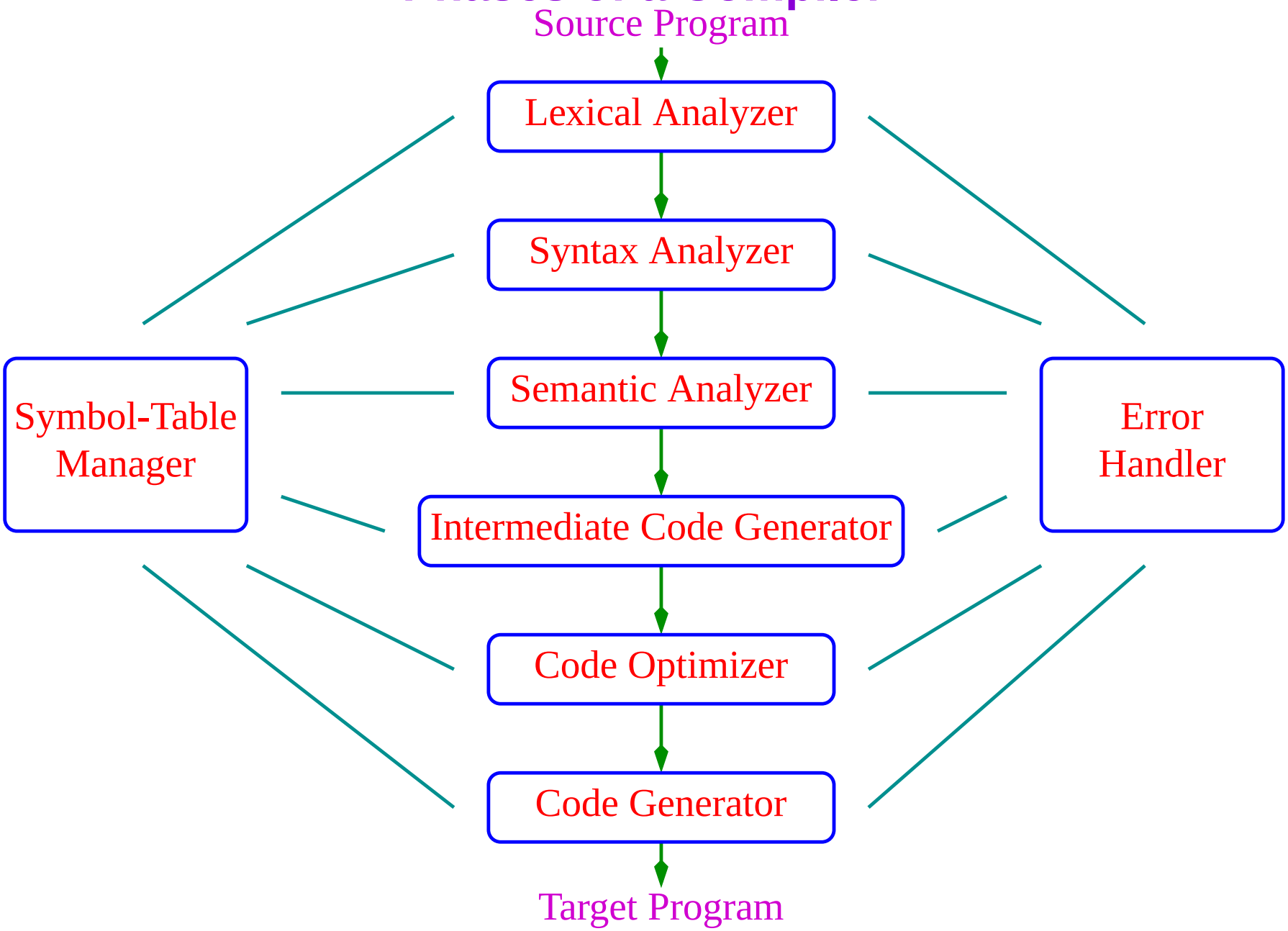
Analysis can be partitioned into three phases:

- Linear (Lexical) Analysis. Stream of characters is read left-to-right and partitioned into tokens.
- Hierarchical (Syntax) Analysis. Tokens are grouped hierarchically into nested collections.

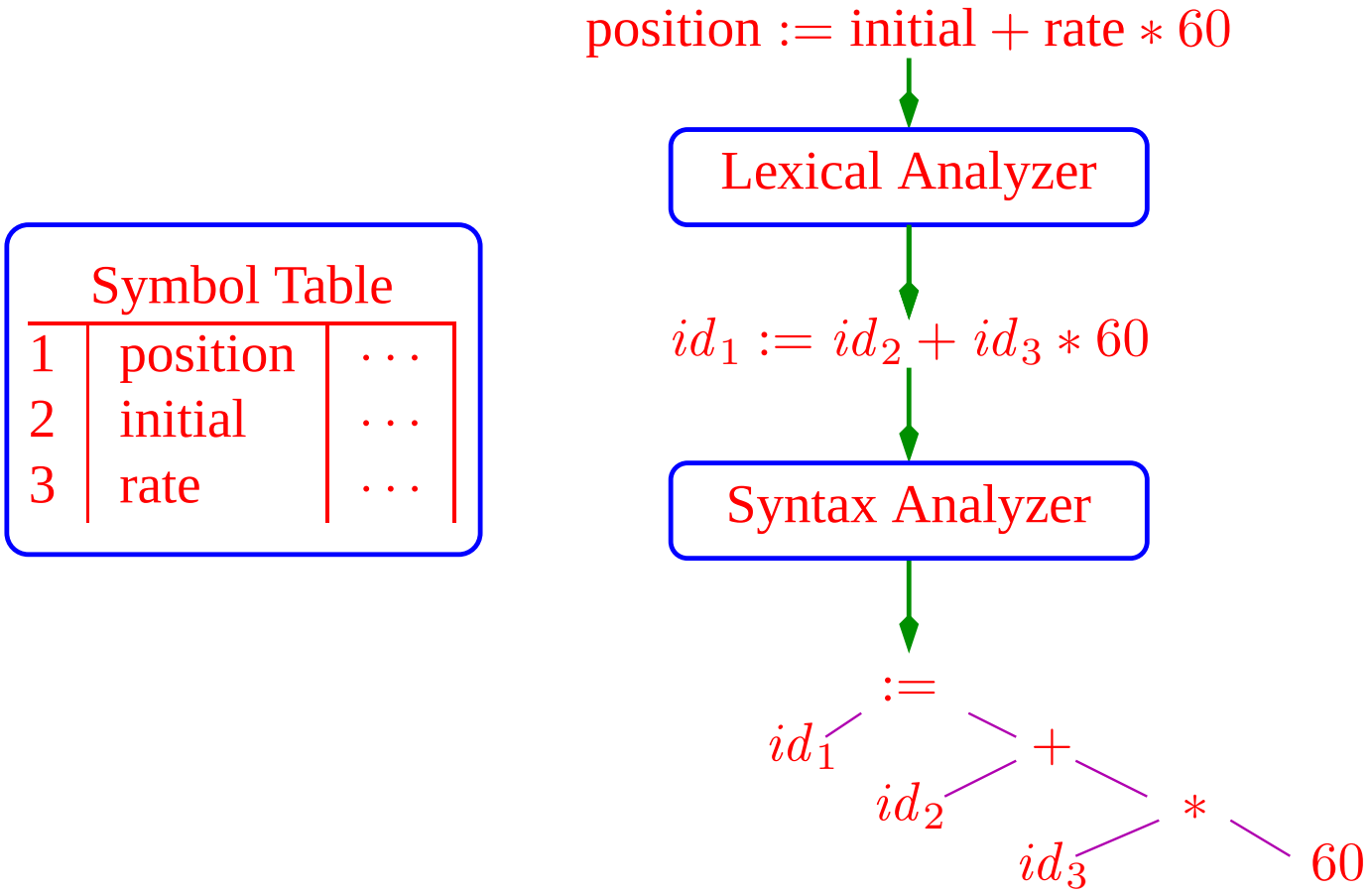


- Semantic Analysis. Checking global consistency. Often does not comply with hierarchical structure. Type checking is an instance of such analysis.

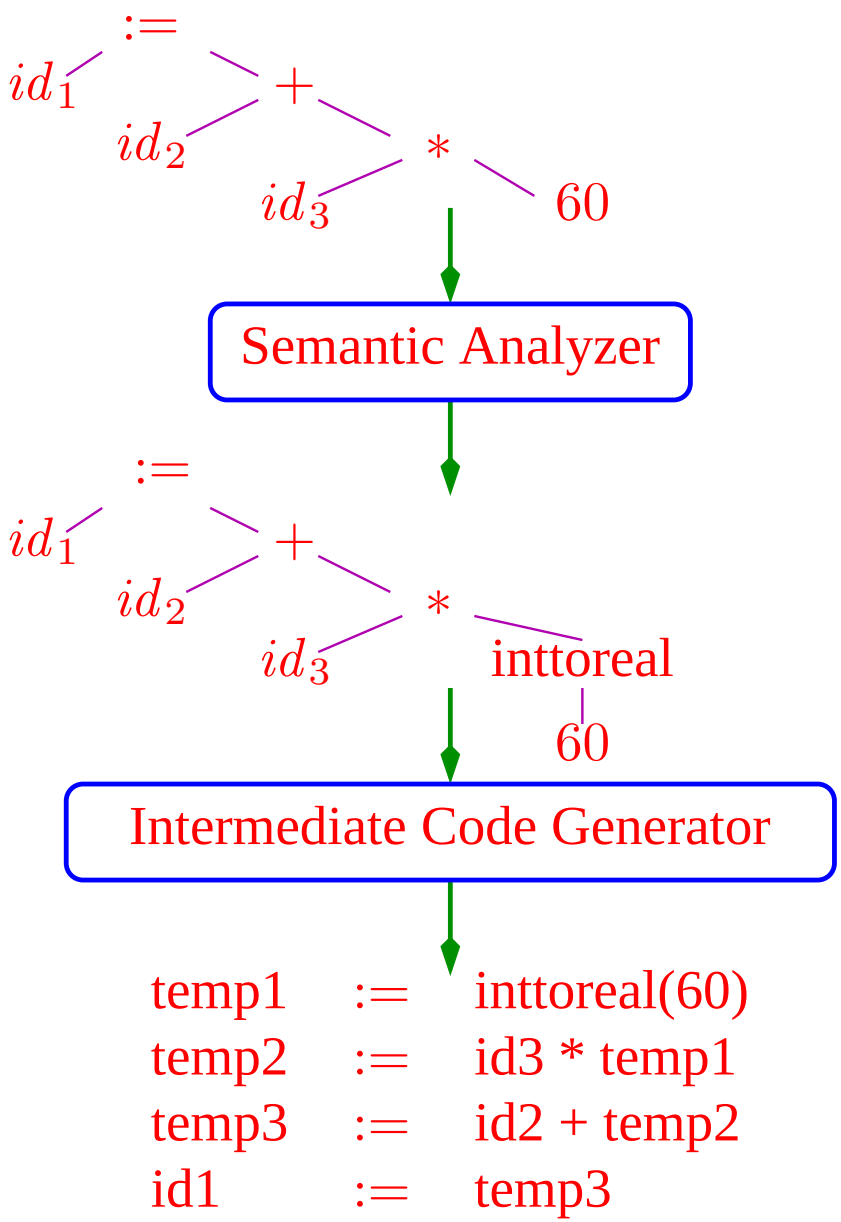
Phases of a Compiler



Illustrate on a Statement

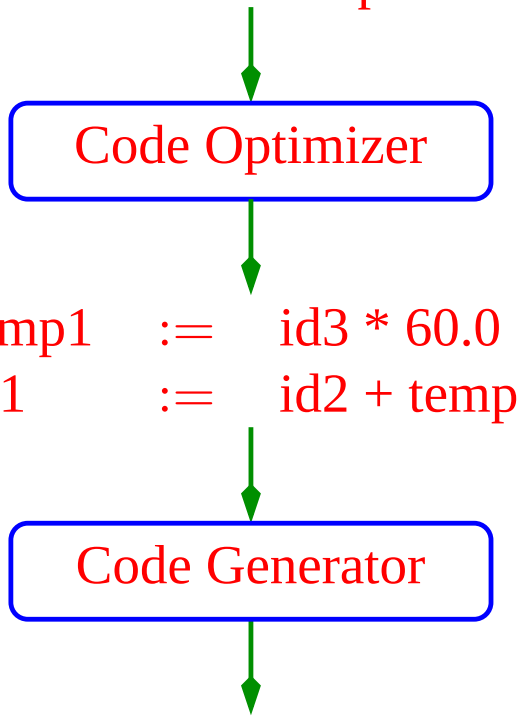


Processing Continued (2/3)



Processing Continued (3/3)

```
temp1   :=   inttoreal(60)
temp2   :=   id3 * temp1
temp3   :=   id2 + temp2
id1     :=   temp3
```




```
temp1   :=   id3 * 60.0
id1     :=   id2 + temp1
```

```
MOVF    id3, R2
MULF    #60.0, R2
MOVF    id2, R1
ADDF    R2, R1
MOVF    R1, id1
```

Assemblers

Many compilers produce symbolic assembly code which is later translated into relocatable code.

A typical assembler proceeds in two passes. The first pass determines addresses of identifiers (relative to the beginning of the program or to the beginning of the data area), placing these addresses in the symbol table.

The second pass translates the code, replacing references to variables by their addresses and placing constants in their right addresses.

For example, the assembler code corresponding to the source statement $b := a + 2$, could be

```
MOV   a,R1
ADD   #2,R1
MOV   R1,b
```

The first pass may decide to allocate a to address $D + 0$ and b to address $D + 4$. The translation could be

```
1   1   0   00   +      Load into R1
3   1   2   02          Add into R1
2   1   0   04   +      Store from R1
```

Syntax Definition

Many definitions of a syntax of a sentence in logic or in a programming language have the following form:

- A number or an identifier is an expression.
- If E_1 and E_2 are expressions, then so are

$$E_1 + E_2, \quad E_1 - E_2, \quad E_1 * E_2, \quad E_1/E_2, \quad (E_1).$$

What is characteristic to such definitions is that they start from some atomic constructs (e.g. number, identifier), introduce one or more categorical constructs (e.g. expression), and then provide rules by which constructs can be combined to yield further instances of constructs.

This mode of definition is generalized to the notion of a context-free grammar (or simply grammar).

Context-Free Grammars

A context-free grammar has four components:

1. A set of tokens, known as terminal symbols.
2. A set of nonterminal symbols (corresponding to categorical concepts).
3. A set of productions of the form $A \rightarrow B_1 \cdots B_k$, where A is a non-terminal, and each B_i is any symbol.
4. A designation of one of the nonterminals as the start symbol.

We often combine the two rules $A \rightarrow B_1 \cdots B_k$ and $A \rightarrow C_1 \cdots C_m$ into $A \rightarrow B_1 \cdots B_k \mid C_1 \cdots C_m$.

Example: List of digits separated by $+$ or $-$.

$$\begin{array}{lcl} \textit{list} & \rightarrow & \textit{list} + \textit{digit} \mid \textit{list} - \textit{digit} \mid \textit{digit} \\ \textit{digit} & \rightarrow & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{array}$$

A grammar derives strings by beginning with the start symbol, and repeatedly replacing a nonterminal symbol by the right side of a production for that terminal. The token (nonterminal) strings that can be derived from the start symbol is the language defined by the grammar.

Example of a Derivation

Given the grammar

$$\begin{array}{lcl} L & \rightarrow & L+D \mid L-D \mid D \\ D & \rightarrow & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{array}$$

we can use it to derive the string $9-5+2$ as follows:

$$\underline{L} \implies \underbrace{\underline{L}+D} \implies \underbrace{\underline{L}-D}+D \implies \underbrace{\underline{D}}-D+D \implies$$

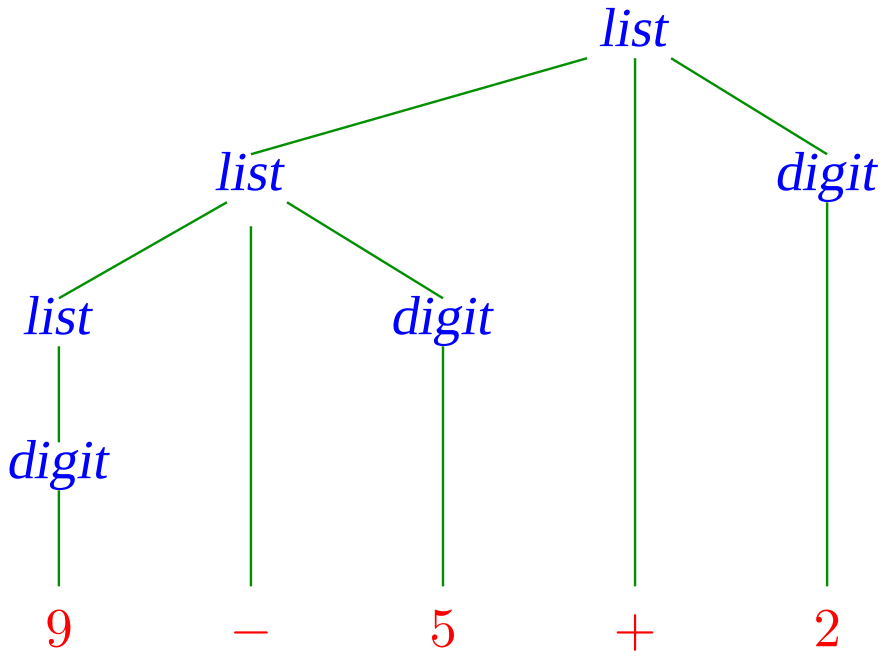
$$\underbrace{9}-\underline{D}+D \implies 9-\underbrace{5}+\underline{D} \implies 9-5+\underbrace{2}$$

The process inverse to derivation is recognition or parsing.

Parse Trees

The history of derivation of a string by a grammar can be represented by a parse tree.

For example, following is the parse tree of the derivation of the string $9 - 5 + 2$ by the grammar for *list*.



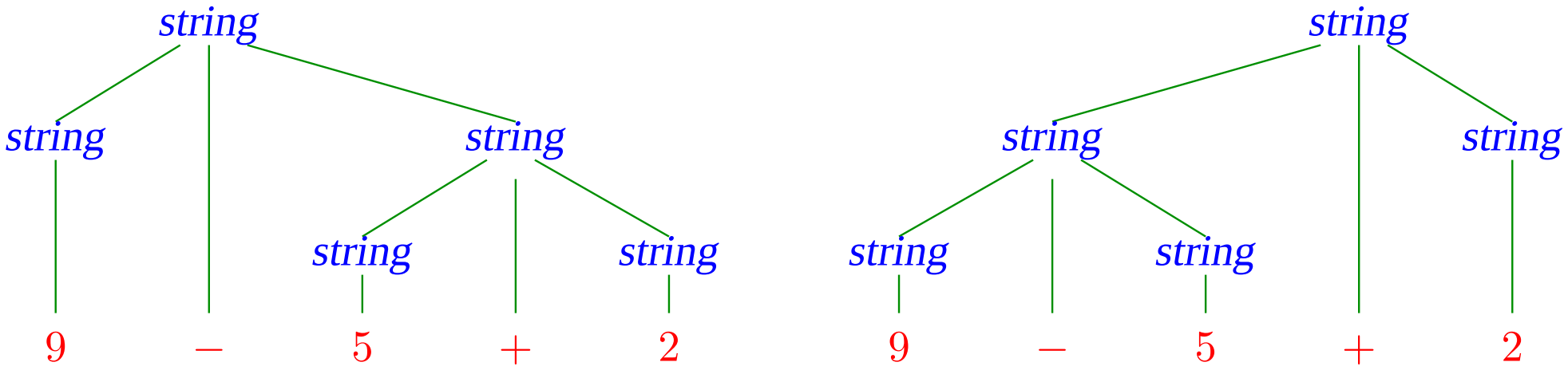
The importance of grammars is not only in their ability to distinguish between acceptable and unacceptable strings. Not less important is the hierarchical grouping they induce on the strings through the parse trees.

Ambiguous Grammars

A grammar is ambiguous if it can produce two different parse trees for the same string.

The grammar for *list* was unambiguous. On the other hand, the following grammar for the same language is ambiguous:

$$\textit{string} \rightarrow \textit{string} + \textit{string} \mid \textit{string} - \textit{string}$$
$$\textit{string} \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$



Among these two parse trees, only the right provides the correct arithmetical grouping. From now on, we will restrict our attention to unambiguous grammars.

Capturing Operator Precedence

Consider the language of parenthesis-free arithmetical expressions over digits and the two operations $+$ and $*$. A possible grammar for this language is

$$\begin{array}{lcl} E & \rightarrow & E + D \mid E * D \mid D \\ D & \rightarrow & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{array}$$

Unfortunately, parsing the string $2 + 4 * 3$ according to this grammar, yields the grouping $\{2 + 4\} * 3$. This grouping implies evaluation of the string to the value 18 instead of the correct value 14.

A grammar that correctly captures the operator precedence is given by:

$$\begin{array}{lcl} E & \rightarrow & E + T \mid T \\ T & \rightarrow & T * D \mid D \\ D & \rightarrow & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{array}$$

Parsing the string $2 + 4 * 3$ according to this two-tiered grammar yields:

$$\overbrace{2 + \overbrace{4 * 3}^{T}}^{E}$$

leading to the value 14.

Easy Reading of Grammars

Consider the extended grammar

$$
\begin{array}{lcl}
E & \rightarrow & E + T \mid E - T \mid T \\
T & \rightarrow & T * D \mid T/D \mid D \\
D & \rightarrow & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{array}
$$

We can interpret it as capturing the following definition:

- An expression (E) is a sequence of terms T separated by the operators $+$ or $-$. $E = T\{+T\}^*$.
- A term (T) is a sequence of digits (D) separated by the operators $*$ or $/$. $T = D\{*D\}^*$.

Capturing Associativity

A grammar such as

$$
\begin{array}{lcl}
E & \rightarrow & E + D \mid D \\
D & \rightarrow & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{array}
$$

Is called left-recursive and captures left associativity. It will parse the string $1 + 2 + 3 + 4$ as

$$\{\{1 + 2\} + 3\} + 4$$

In contrast, the grammar

$$
\begin{array}{lcl}
A & \rightarrow & id := A \mid id := D \\
id & \rightarrow & a \mid \cdots \mid z \\
D & \rightarrow & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{array}
$$

Is called right-recursive and captures right associativity. It will parse the string $a := b := c := d := 5$ as

$$a := \{b := \{c := \{d := 5\}\}\}$$