2.2 A Simple Syntax-Directed Translator
Note for the students:

• These slides are meant for the lecturers to conduct lectures only. It is NOT suitable to be used as a study material.
• Students are expected to study by reading the textbook for this course:
• 2.3 Syntax-Directed Translation
• 2.4 Parsing
• 2.5 A Translator for Simple Expressions
• 2.6 Lexical Analysis
• 2.7 Symbol Tables
• 2.8 Intermediate Code Generation
• 2.9 Summary of Chapter 2
Syntax-Directed Translation

• Syntax-directed translation is done by attaching rules or program fragments to productions in a grammar.

• For example, consider an expression $expr$ generated by the production:

$$expr \rightarrow expr_1 + term$$

• We can translate $expr$ by exploiting its structure, as in the following pseudo-code:

```plaintext
translate expr_1;
translate term;
handle +;
```
Concepts of syntax-directed translation

• Attributes.
  – An attribute is any quantity associated with a programming construct. Examples of attributes are data types of expressions, the number of instructions in the generated code, or the location of the first instruction in the generated code for a construct, among many other possibilities.

• (Syntax-directed) translation schemes.
  – A translation scheme is a notation for attaching program fragments to the productions of a grammar. The program fragments are executed when the production is used during syntax analysis.
Postfix Notation

• The postfix notation for an expression $E$ can be defined inductively as follows:

1. If $E$ is a variable or constant, then the postfix notation for $E$ is $E$ itself.

2. If $E$ is an expression of the form $E_1 \, op \, E_2$, where $op$ is any binary operator, then the postfix notation for $E$ is $E'_1 \, E'_2 \, op$, where $E'_1$ and $E'_2$ are the postfix notations for $E_1$ and $E_2$, respectively.

3. If $E$ is a parenthesized expression of the form $(E_1)$, then the postfix notation for $E$ is the same as the postfix notation for $E_1$.

• Example 2.8 : The postfix notation for $(9-5)+2$ is 95-2+.
• Example 2.9 : (try)
Synthesized Attributes

• We associate attributes with nonterminals and terminals. Then, we attach rules to the productions of the grammar; these rules describe how the attributes are computed at those nodes of the parse tree where the production in question is used to relate a node to its children.
Figure 2.9: Attribute values at nodes in a parse tree

Suppose a node $N$ in a parse tree is labeled by the grammar symbol $X$. We write $X.a$ to denote the value of attribute $a$ of $X$ at that node. A parse tree showing the attribute values at each node is called an annotated parse tree. For example, Fig. 2.9 shows an annotated parse tree for $9-5+2$ with an attribute $t$ associated with the nonterminals $expr$ and $term$. The value $95-2+$ of the attribute at the root is the postfix notation for $9-5+2$. We shall see shortly how these expressions are computed.
Example 2.10: The annotated parse tree in Fig. 2.9 is based on the syntax-directed definition in Fig. 2.10 for translating expressions consisting of digits separated by plus or minus signs into postfix notation. Each nonterminal has a string-valued attribute \( t \) that represents the postfix notation for the expression generated by that nonterminal in a parse tree. The symbol || in the semantic rule is the operator for string concatenation.

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>( expr \to expr_1 + term )</td>
<td>( expr.t = expr_1.t</td>
</tr>
<tr>
<td>( expr \to expr_1 - term )</td>
<td>( expr.t = expr_1.t</td>
</tr>
<tr>
<td>( expr \to term )</td>
<td>( expr.t = term.t )</td>
</tr>
<tr>
<td>( term \to 0 )</td>
<td>( term.t = '0' )</td>
</tr>
<tr>
<td>( term \to 1 )</td>
<td>( term.t = '1' )</td>
</tr>
<tr>
<td>( ... )</td>
<td>( ... )</td>
</tr>
<tr>
<td>( term \to 9 )</td>
<td>( term.t = '9' )</td>
</tr>
</tbody>
</table>

Figure 2.10: Syntax-directed definition for infix to postfix translation
Tree Traversals

• Tree traversals will be used for describing attribute evaluation and for specifying the execution of code fragments in a translation scheme.
  – A traversal of a tree starts at the root and visits each node of the tree in some order.
  – A depth-first traversal starts at the root and recursively visits the children of each node in any order, not necessarily from left to right. It is called "depth first" because it visits an unvisited child of a node whenever it can, so it visits nodes as far away from the root (as "deep") as quickly as it can.
procedure visit(node N) {
    for (each child C of N, from left to right) {
        visit(C);
    }
    evaluate semantic rules at node N;
}

Figure 2.11: A depth-first traversal of a tree

Figure 2.12: Example of a depth-first traversal of a tree
Translation Schemes

• Preorder and Postorder Traversals
Semantic actions

\[ rest \rightarrow + \ term \ \{\text{print('+' \})\} \ rest_1 \]

Figure 2.13: An extra leaf is constructed for a semantic action
Parsing

• Parsing is the process of determining how a string of terminals can be generated by a grammar.
  – Top-Down Parsing

The top-down construction of a parse tree like the one in Fig. 2.17, is done by starting with the root, labeled with the starting nonterminal *stmt*, and repeatedly performing the following two steps.

1. At node N, labeled with nonterminal A, select one of the productions for A and construct children at N for the symbols in the production body.

2. Find the next node at which a subtree is to be constructed, typically the leftmost unexpanded nonterminal of the tree.
\[ stmt \rightarrow \text{expr ;} \\
\quad \mid \text{if ( expr ) stmt} \\
\quad \mid \text{for ( optexpr ; optexpr ; optexpr ) stmt} \\
\quad \mid \text{other} \]

\[ optexpr \rightarrow \epsilon \\
\quad \mid \text{expr} \]

Figure 2.16: A grammar for some statements in C and Java

Figure 2.17: A parse tree according to the grammar in Fig. 2.16
Use \( \epsilon \)-Productions

- Predictive Parsing (self study)
- When to Use \( \epsilon \)-Productions (Our predictive parser uses an \( \epsilon \)-production as a default when no other production can be used)
Figure 2.20: Left- and right-recursive ways of generating a string
A Translator for Simple Expressions

• Using the techniques of the last three sections, we now construct a syntax directed translator
• A syntax-directed translation scheme often serves as the specification for a translator. The scheme in Fig. 2.21 (repeated from Fig. 2.15) defines the translation to be performed her

```
expr  →  expr + term  { print('+') }
       |        |        |
       expr - term { print('-') }
       |        |        |
       term    |

term  →  0      { print('0') }
       | 1      { print('1') }
       | ...    |
       | 9      { print('9') }
```

Figure 2.21: Actions for translating into postfix notation
Abstract and Concrete Syntax

- A useful starting point for designing a translator is a data structure called an abstract syntax tree.
- In an abstract syntax tree for an expression, each interior node represents an operator; the children of the node represent the operands of the operator.

In the abstract syntax tree for $9-5+2$ in Fig. 2.22, the root represents the operator $+$. The subtrees of the root represent the subexpressions $9-5$ and $2$. The grouping of $9-5$ as an operand reflects the left-to-right evaluation of operators at the same precedence level. Since $-$ and $+$ have the same precedence, $9-5+2$ is equivalent to $(9-5)+2$. 
Figure 2.41: Concrete and abstract syntax for several Java operators
syntax tree vs. parse tree

- in the syntax tree, interior nodes represent programming constructs while in the parse tree, the interior nodes represent nonterminals.
\[
expr \rightarrow \text{term rest}
\]
\[
\text{rest} \rightarrow + \text{term} \{ \text{print('i')} \} \text{rest} \\
| - \text{term} \{ \text{print('j')} \} \text{rest} \\
| \epsilon
\]
\[
\text{term} \rightarrow 0 \{ \text{print('0')} \} \\
| 1 \{ \text{print('1')} \} \\
| \ldots \\
| 9 \{ \text{print('9')} \}
\]

Figure 2.23: Translation scheme after left-recursion elimination

\[
\text{expr} \\
| \text{term} \\
| \text{rest}
\]
\[
\text{term} \\
| 9 \{ \text{print('9')} \}
\]
\[
\text{rest} \\
| - \text{term} \{ \text{print('j')} \} \text{rest}
\]
\[
\text{term} \\
| 5 \{ \text{print('5')} \}
\]
\[
\text{rest} \\
| + \text{term} \{ \text{print('i')} \} \text{rest}
\]
\[
\text{term} \\
| 2 \{ \text{print('2')} \}
\]
\[
\epsilon
\]

Figure 2.24: Translation of 9-5+2 to 95-2+
Lexical Analysis

• A lexical analyzer reads characters from the input and groups them into "token objects."
• The lexical analyzer in this section allows numbers, identifiers, and "white space" (blanks, tabs, and newlines) to appear within expressions.

```
expr  →  expr + term  { print('+') }
     |  expr - term  { print('-') }
     |  term

term  →  term * factor  { print('*') }
       |  term / factor  { print('/') }
       |  factor

factor  →  ( expr )  { print(expr.lexeme) }
        |  num  { print(num.value) }
        |  id  { print(id.lexeme) }
```

Figure 2.28: Actions for translating into postfix notation
Reading Ahead

• A lexical analyzer may need to read ahead some characters before it can decide on the token to be returned to the parser.

• For example, a lexical analyzer for C or Java must read ahead after it sees the character >.
  – If the next character is =, then > is part of the character sequence >=, the lexeme for the token for the "greater than or equal to" operator.
  – Otherwise > itself forms the "greater than" operator, and the lexical analyzer has read one character too many.
When a sequence of digits appears in the input stream, the lexical analyzer passes to the parser a token consisting of the terminal `num` along with an integer-valued attribute computed from the digits. If we write tokens as tuples enclosed between ⟨ ⟩, the input 31 + 28 + 59 is transformed into the sequence

⟨num,31⟩ ⟨+⟩ ⟨num,28⟩ ⟨+⟩ ⟨num,59⟩

Here, the terminal symbol `+` has no attributes, so its tuple is simply ⟨+⟩.
Recognizing Keywords and Identifiers

Most languages use fixed character strings such as for, do, and if, as punctuation marks or to identify constructs. Such character strings are called keywords.

Character strings are also used as identifiers to name variables, arrays, functions, and the like. Grammars routinely treat identifiers as terminals to simplify the parser, which can then expect the same terminal, say id, each time any identifier appears in the input. For example, on input

\[
\text{count} = \text{count} + \text{increment};
\]

the parser works with the terminal stream \( \text{id} = \text{id} + \text{id} \). The token for \( \text{id} \) has an attribute that holds the lexeme. Writing tokens as tuples, we see that the tuples for the input stream (2.6) are

\[
\langle \text{id}, "\text{count}" \rangle \ (=) \langle \text{id}, "\text{count}" \rangle \ (+) \langle \text{id}, "\text{increment}" \rangle \langle ; \rangle
\]

Keywords generally satisfy the rules for forming identifiers, so a mechanism is needed for deciding when a lexeme forms a keyword and when it forms an identifier. The problem is easier to resolve if keywords are reserved; i.e., if they cannot be used as identifiers. Then, a character string forms an identifier only if it is not a keyword.
Symbol table

• Symbol tables are data structures that are used by compilers to hold information about source-program constructs.
• The information is collected incrementally by the analysis phases of a compiler and used by the synthesis phases to generate the target code.
• Entries in the symbol table contain information about an identifier such as its character string (or lexeme), its type, its position in storage, and any other relevant information. Symbol tables typically need to support multiple declarations of the same identifier within a program.
Scope

• Symbol Table Per Scope: The term "scope of identifier x' really refers to the scope of a particular declaration of x.

• Scopes are important, because the same identifier can be declared for different purposes in different parts of a program.

• If blocks can be nested, several declarations of the same identifier can appear within a single block.
Example 2.15: The following pseudocode uses subscripts to distinguish among distinct declarations of the same identifier:

1) \{ \text{int } x_1; \text{ int } y_1; \\
2) \quad \{ \text{int } w_2; \text{ bool } y_2; \text{ int } z_2; \\
3) \quad \quad \cdots \text{ } w_2 \cdots; \cdots \text{ } x_1 \cdots; \cdots \text{ } y_2 \cdots; \cdots \text{ } z_2 \cdots; \\
4) \quad \} \\
5) \quad \cdots \text{ } w_0 \cdots; \cdots \text{ } x_1 \cdots; \cdots \text{ } y_1 \cdots; \\
6) \}  \\

Figure 2.36: Chained symbol tables for Example 2.15
Intermediate Code Generation

• Two Kinds of Intermediate Representations

  • Trees, including parse trees and (abstract) syntax trees.
  • Linear representations, especially “three-address code.”
Construction of Syntax Trees

Figure 2.39: Construction of syntax trees for expressions and statements
L-values and R-values

• There is a distinction between the meaning of identifiers on the left and right sides of an assignment. In each of the assignments the right side specifies an integer value, while the left side specifies where the value is to be stored.

\[
\begin{align*}
i &= 5; \\
i &= i + 1;
\end{align*}
\]
Type Checking

```plaintext
if ( expr ) stmt
```

```plaintext
if ( E1.type == E2.type ) E.type = boolean;
else error;
```
Three-Address Code

Three-address code is a sequence of instructions of the form

\[ x = y \text{ op } z \]

where \( x, y, \) and \( z \) are names, constants, or compiler-generated temporaries; and \( \text{op} \) stands for an operator.

Arrays will be handled by using the following two variants of instructions:

\[ x [ y ] = z \]
\[ x = y [ z ] \]
Example 2.20: When applied to the syntax tree for

\[ a[i] = 2*a[j-k] \]

function \textit{rvalue} generates

\[ t3 = j - k \]
\[ t2 = a[t3] \]
\[ t1 = 2 \times t2 \]
\[ a[i] = t1 \]
Summary of Chapter 2

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

**Lexical Analyzer**

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

**Syntax-Directed Translator**

**Figure 2.46: Two possible translations of a statement**

1: t1 = (int) '\n'
2: ifFalse peek == t1 goto 4
3: line = line + 1
4: