بسم اللّه الرّحمن الرّحیم

دانشگاه صنعتی اصفهان ـ دانشکدهٔ مهندسی برق و کامپیوتر
(نیم‌سال تحصیلی ۴۰۱۲)

کامپایلر

حسین فلسفین

Chapter 2: A Simple Syntax-Directed Translator

راهی که برای بدست آوردن کد میانی پیش رو داریم را قرار است به اجمال طی کنیم

Chapter 2 is an introduction to the compiling techniques in Chapters 3 through 6. It illustrates the techniques by developing a working Java program that translates representative programming language statements into three-address code, an intermediate representation. In Chapter 2, the emphasis is on the front end of a compiler, in particular on lexical analysis, parsing, and intermediate code generation. Chapters 7 and 8 show how to generate machine instructions from three-address code.

We start small by creating a syntax-directed translator that maps infix arithmetic expressions into postfix expressions. We then extend this translator to map code fragments as shown in Fig. 2.1 into three-address code of the form in Fig. 2.2.

```
{
    int i; int j; float[100] a; float v; float x;
    while ( true ) {
        do i = i+1; while ( a[i] < v );
        do j = j-1; while ( a[j] > v );
        if ( i >= j ) break;
        x = a[i]; a[i] = a[j]; a[j] = x;
    }
}
```

Figure 2.1: A code fragment to be translated

```
 1:  i = i + 1
 2:  t1 = a [ i ]
 3:  if t1 < v goto 1
 4:  j = j - 1
 5:  t2 = a [ j ]
 6:  if t2 > v goto 4
 7:  ifFalse i >= j goto 9
 8:  goto 14
 9:  x = a [ i ]
10:  t3 = a [ j ]
11:  a [ i ] = t3
12:  a [ j ] = x
13:  goto 1
14:
```

Figure 2.2: Simplified intermediate code for the program fragment in Fig. 2.1

فعلاً قصد داریم تا دیدی کلی و اجمالی نسبت به فاز آنالیز کامپایلر داشته باشیم

The analysis phase of a compiler breaks up a source program into constituent pieces and produces an internal representation for it, called intermediate code. The synthesis phase translates the intermediate code into the target program.

Analysis is organized around the “syntax” of the language to be compiled. The syntax of a programming language describes the proper form of its programs, while the semantics of the language defines what its programs mean; that is, what each program does when it executes.

☞ *For specifying syntax, we present a widely used notation, called context-free grammars or BNF (for Backus-Naur Form).*

☞ *With the notations currently available, the semantics of a language is much more difficult to describe than the syntax. For specifying semantics, we shall therefore use informal descriptions and suggestive examples.*

Syntax-Directed Translation

Besides specifying the syntax of a language, a context-free grammar can be used to help guide the translation of programs. In Section 2.3, we introduce a grammar-oriented compiling technique known as syntax-directed translation.

In Chapter 6 (entitled intermediate-code generation), the syntax-directed formalisms of Chapters 2 and 5 are used to specify checking and translation.

Chapter 2 is a quick tour through the model of a compiler front end in Fig. 2.3.

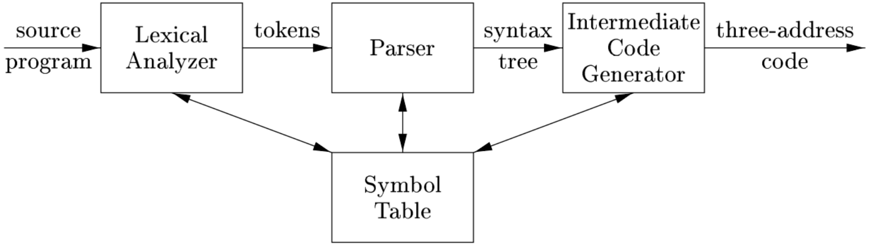


Figure 2.3: A model of a compiler front end

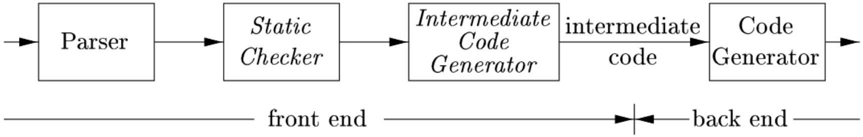
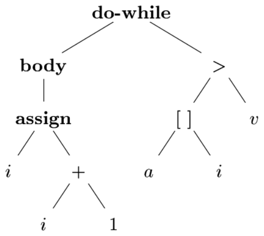


Figure 6.1: Logical structure of a compiler front end

Chapter 2 also considers intermediate-code generation. Two forms of intermediate code are illustrated in Fig. 2.4. One form, called abstract syntax trees or simply syntax trees, represents the hierarchical syntactic structure of the source program. In the model in Fig. 2.3, the parser produces a syntax tree, that is further translated into three-address code. Some compilers combine parsing and intermediate-code generation into one component.



(a)

```
1: i = i + 1
2: t1 = a [ i ]
3: if t1 < v goto 1
```

(b)

Figure 2.4: Intermediate code for "`do i = i + 1; while ( a[i] < v );`"

☞ *The root of the abstract syntax tree in Fig. 2.4(a) represents an entire do-while loop. The left child of the root represents the body of the loop, which consists of only the assignment* `i=i+1;`*. The right child of the root represents the condition* `a[i]<v`*.*

☞ *The other common intermediate representation, shown in Fig. 2.4(b), is a sequence of "three-address" instructions; a more complete example appears in Fig. 2.2. This form of intermediate code takes its name from instructions of the form* $x = y \, \mathrm{op} \, z$*, where* op *is a binary operator,* y *and* z *are the addresses for the operands, and* x *is the address for the result of the operation. A three-address instruction carries out at most one operation, typically a computation, a comparison, or a branch.*

2.3 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 $expr \to expr_1 + term$ Here, $expr$ is the sum of the two subexpressions $expr_1$ and $term$. (The subscript in $expr_1$ is used only to distinguish the instance of $expr$ in the production body from the head of the production). We can translate $expr$ by exploiting its structure, as in the following pseudo-code:

```
translate $expr_1$;   translate $term$;   handle $+$;
```

Using a variant of this pseudocode, we shall build a syntax tree for $expr$ in Section 2.8 by building syntax trees for $expr_1$ and $term$ and then handling $+$ by constructing a node for it. For convenience, the example in this section is the translation of infix expressions into postfix notation.

Syntax-Directed Translation: the translation of languages guided by CFGs

The translation techniques in Chapter 5 will be applied in Chapter 6 to type checking and intermediate-code generation. The techniques are also useful for implementing little languages for specialized tasks.

We associate information with a language construct by attaching attributes to the grammar symbol(s) representing the construct.

Two concepts related to 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. Since we use grammar symbols (nonterminals and terminals) to represent programming constructs, we extend the notion of attributes from constructs to the symbols that represent them.*

☞ *(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. The combined result of all these fragment executions, in the order induced by the syntax analysis, produces the translation of the program to which this analysis/synthesis process is applied.*

Syntax-directed translations will be used throughout this chapter (i.e., Chapter 2) to translate infix expressions into postfix notation, to evaluate expressions, and to build syntax trees for programming constructs. A more detailed discussion of syntax-directed formalisms appears in Chapter 5.

2.3.1 Postfix Notation

The examples in this section deal with translation into 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+`. That is, the translations of `9`, `5`, and `2` are the constants themselves, by rule (1). Then, the translation of `9-5` is `95-` by rule (2). The translation of `(9-5)` is the same by rule (3). Having translated the parenthesized subexpression, we may apply rule (2) to the entire expression, with `(9-5)` in the role of E_1 and `2` in the role of E_2, to get the result `95-2+`.

As another example, the postfix notation for `9-(5+2)` is `952+-`. That is, `5+2` is first translated into `52+`, and this expression becomes the second argument of the minus sign. □

No parentheses are needed in postfix notation, because the position and arity (number of arguments) of the operators permits only one decoding of a postfix expression. The “trick” is to repeatedly scan the postfix string from the left, until you find an operator. Then, look to the left for the proper number of operands, and group this operator with its operands. Evaluate the operator on the operands, and replace them by the result. Then repeat the process, continuing to the right and searching for another operator.

Example 2.9: Consider the postfix expression 952+−3∗. *Scanning from the left, we first encounter the plus sign. Looking to its left we find operands* 5 *and* 2. *Their sum,* 7, *replaces* 52+, *and we have the string* 97−3∗. *Now, the leftmost operator is the minus sign, and its operands are* 9 *and* 7. *Replacing these by the result of the subtraction leaves* 23∗. *Last, the multiplication sign applies to* 2 *and* 3, *giving the result* 6.

2.3.2 Synthesized Attributes

The idea of associating quantities with programming constructs—for example, values and types with expressions—can be expressed in terms of grammars. We associate attributes with non-terminals 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.

Syntax-Directed Definition (SDDs)

A syntax-directed definition (SDD) is a context-free grammar together with attributes and rules. Attributes are associated with grammar symbols and rules are associated with productions. If X is a symbol and a is one of its attributes, then we write $X.a$ to denote the value of a at a particular parse-tree node labeled X. If we implement the nodes of the parse tree by records or objects, then the attributes of X can be implemented by data fields in the records that represent the nodes for X. Attributes may be of any kind: numbers, types, table references, or strings, for instance. The strings may even be long sequences of code, say code in the intermediate language used by a compiler.

A syntax-directed definition associates
1. With each grammar symbol, a set of attributes, and
2. With each production, a set of semantic rules for computing the values of the attributes associated with the symbols appearing in the production.

Attributes can be evaluated as follows. For a given input string x, construct a parse tree for x. Then, apply the semantic rules to evaluate attributes at each node in the parse tree, as follows.

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.*

مثالی از بکارگیری *SDD*ها برای مقصود ترجمه

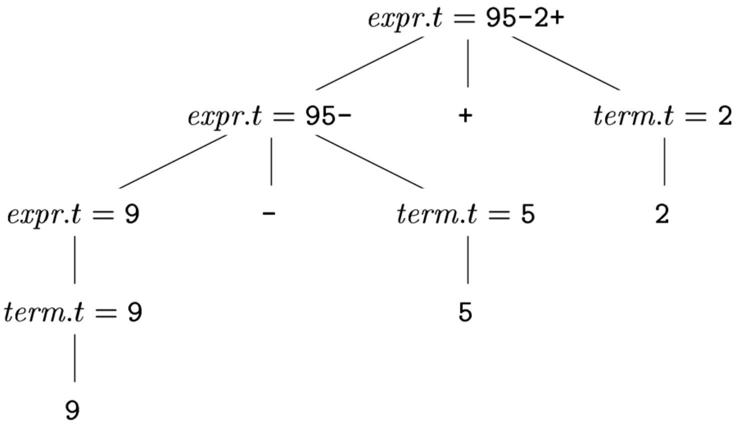


Figure 2.9: Attribute values at nodes in a parse tree

مثالی از بکارگیری *SDD*ها برای مقصود ترجمه

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.

PRODUCTION	SEMANTIC RULES
$expr \rightarrow expr_1 + term$	$expr.t = expr_1.t \,\|\|\, term.t \,\|\|\, '+'$
$expr \rightarrow expr_1 - term$	$expr.t = expr_1.t \,\|\|\, term.t \,\|\|\, '-'$
$expr \rightarrow term$	$expr.t = term.t$
$term \rightarrow 0$	$term.t = '0'$
$term \rightarrow 1$	$term.t = '1'$
$\cdots$	$\cdots$
$term \rightarrow 9$	$term.t = '9'$

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

The postfix form of a digit is the digit itself; e.g., the semantic rule associated with the production $term \rightarrow 9$ defines $term.t$ to be 9 itself whenever this production is used at a node in a parse tree. The other digits are translated similarly. As another example, when the production $expr \rightarrow term$ is applied, the value of $term.t$ becomes the value of $expr.t$.

The production $expr \rightarrow expr_1 + term$ derives an expression containing a plus operator. The left operand of the plus operator is given by $expr_1$ and the right operand by $term$. The semantic rule

$$expr.t \;=\; expr_1.t \;||\; term.t \;||\; '+'$$

associated with this production constructs the value of attribute $expr.t$ by concatenating the postfix forms $expr_1.t$ and $term.t$ of the left and right operands, respectively, and then appending the plus sign. This rule is a formalization of the definition of "postfix expression." $\square$

A syntax-directed definition specifies the values of attributes by associating semantic rules with the grammar productions. For example, an infix-to-postfix translator might have a production and rule

PRODUCTION	SEMANTIC RULE	
$E \rightarrow E_1 + T$	$E.code = E_1.code \parallel T.code \parallel '+'$	(5.1)

This production has two nonterminals, E and T; the subscript in E_1 distinguishes the occurrence of E in the production body from the occurrence of E as the head. Both E and T have a string-valued attribute *code*. The semantic rule specifies that the string $E.code$ is formed by concatenating $E_1.code$, $T.code$, and the character '+'. While the rule makes it explicit that the translation of E is built up from the translations of E_1, T, and '+', it may be inefficient to implement the translation directly by manipulating strings.

Inherited and Synthesized Attributes

An attribute is said to be synthesized if its value at a parse-tree node N is determined from attribute values at the children of N and at N itself. Synthesized attributes have the desirable property that they can be evaluated during a single bottom-up traversal of a parse tree. In Section 5.1.1 we shall discuss another important kind of attribute: the "inherited" attribute. Informally, inherited attributes have their value at a parse-tree node determined from attribute values at the node itself, its parent, and its siblings in the parse tree.

2.3.3 Simple Syntax-Directed Definitions

The syntax-directed definition in Example 2.10 has the following important property: the string representing the translation of the nonterminal at the head of each production is the concatenation of the translations of the nonterminals in the production body, in the same order as in the production, with some optional additional strings interleaved. A syntax-directed definition with this property is termed *simple*.

Example 2.11 : Consider the first production and semantic rule from Fig. 2.10:

$$\begin{array}{ll} \textsc{Production} & \textsc{Semantic Rule} \\ expr \rightarrow expr_1 + term & expr.t = expr_1.t \;||\; term.t \;||\; '+' \end{array} \tag{2.5}$$

Here the translation *expr.t* is the concatenation of the translations of $expr_1$ and *term*, followed by the symbol +. Notice that $expr_1$ and *term* appear in the same order in both the production body and the semantic rule. There are no additional symbols before or between their translations. In this example, the only extra symbol occurs at the end. □

When translation schemes are discussed, we shall see that a simple syntax-directed definition can be implemented by printing only the additional strings, in the order they appear in the definition.