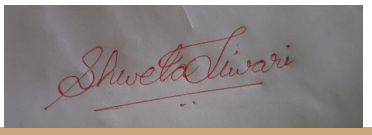
CD: COMPILER DESIGN
CD: UNIT-3 09/2022

SEPTEMBER 2022 / IT-3rd year, Vth semester
FALL SEMESTER, YEAR (V/VI, 3rd)
FALL SESSION (2022-23)
(CD)
MS. SHWETA TIWARI
Published: SEPTEMBER, 2022

PREPARED FOR
Engineering Students
All Engineering College



TOPIC On : UNIT-3 Implementing L-Attributed SDD's

—

By SHWETA TIWARI

Under On: Syntax-Directed Translation

1 Syntax-Directed Definitions

2 Evaluation Orders for SDD's

3 Applications of Syntax-Directed Translation

4 Syntax-Directed Translation Schemes

5 Implementing L-Attributed SDD's

TOPIC On : UNIT-3
Implementing L-Attributed SDD's

Implementing L-Attributed SDD's

Since many translation applications can be addressed using L-attributed defi-nitions, we shall consider their implementation in more detail in this section. The following methods do translation by traversing a parse tree:

1.Build the parse tree and annotate. This method works for any noncircular SDD whatsoever. We introduced annotated parse trees in Section 5.1.2.

2. *Build the parse tree, add actions, and execute the actions in preorder.*

This approach works for any L-attributed definition. We discussed how to turn an L-attributed SDD into an SDT in Section 5.4.5; in particular, we discussed how to embed actions into productions based on the semantic rules of such an SDD.

In this section, we discuss the following methods for translation during parsing:

3. Use a recursive-descent parser with one function for each nonterminal. The function for nonterminal A receives the inherited attributes of A as arguments and returns the synthesized attributes of A.

4. Generate code on the fly, using a recursive-descent parser.

5. Implement an SDT in conjunction with an LL-parser. The attributes are kept on the parsing stack, and the rules fetch the needed attributes from known locations on the stack.

6. Implement an SDT in conjunction with an LR-parser. This method may be surprising, since the SDT for an L-attributed SDD typically has ac-tions in the middle of productions, and we cannot be sure during an LR parse that we are even in that production until its entire body has been constructed. We shall see, however, that if the underlying grammar is LL, we can always handle both the parsing and translation bottom-up.

In this section, we discuss the following methods for translation during parsing:

3. Use a recursive-descent parser with one function for each nonterminal. The function for nonterminal A receives the inherited attributes of A as arguments and returns the synthesized attributes of A.

4. Generate code on the fly, using a recursive-descent parser.

5. Implement an SDT in conjunction with an LL-parser. The attributes are kept on the parsing stack, and the rules fetch the needed attributes from known locations on the stack.

6. Implement an SDT in conjunction with an LR-parser. This method may be surprising, since the SDT for an L-attributed SDD typically has ac-tions in the middle of productions, and we cannot be sure during an LR parse that we are even in that production until its entire body has been constructed. We shall see, however, that if the underlying grammar is LL, we can always handle both the parsing and translation bottom-up.

1. Translation During Recursive-Descent Parsing

A recursive-descent parser has a function A for each nonterminal A, as discussed in Section 4.4.1. We can extend the parser into a translator as follows:

a) The arguments of function A are the inherited attributes of nonterminal A.

b) The return-value of function A is the collection of synthesized attributes of nonterminal A.

In the body of function A, we need to both parse and handle attributes:

Decide upon the production used to expand A.

Check that each terminal appears on the input when it is required. We shall assume that no backtracking is needed, but the extension to recur-sive-descent parsing with backtracking can be done by restoring the input position upon failure, as discussed in Section 4.4.1.

Preserve, in local variables, the values of all attributes needed to compute inherited attributes for nonterminals in the body or synthesized attributes for the head nonterminal.

Call functions corresponding to nonterminals in the body of the selected production, providing them with the proper arguments. Since the un-derlying SDD is L-attributed, we have already computed these attributes and stored them in local variables.

Example 5.20 : Let us consider the SDD and SDT of Example 5.19 for while-statements. A pseudocode rendition of the relevant parts of the function S appears in Fig. 5.29.

```
string S(label next) {
    string Scode, Ccode; /* local variables holding code fragments */
    label L1, L2; /* the local labels */
    if ( current input == token while ) {
        advance input;
        check '(' is next on the input, and advance;
        L1 = new();
        L2 = new();
        Ccode = C(next, L2);
        check ')' is next on the input, and advance;
        Scode = S(L1);
        return("label" || L1 || Ccode || "label" || L2 || Scode);
    }
    else /* other statement types */
}
```

Figure 5.29: Implementing while-statements with a recursive-descent parser

We show S as storing and returning long strings. In practice, it would be far more efficient for functions like S and C to return pointers to records that represent these strings. Then, the return-statement in function S would not physically concatenate the components shown, but rather would construct a record, or perhaps tree of records, expressing the concatenation of the strings represented by Scode and Ccode, the labels LI and L2, and the two occurrences of the literal string "label". •

Example 5.21 : Now, let us take up the SDT of Fig. 5.26 for typesetting boxes. First, we address parsing, since the underlying grammar in Fig. 5.26 is ambiguous. The following transformed grammar makes juxtaposition and subscripting right associative, with sub taking precedence over juxtaposition:

$$
\begin{array}{lcl}
S & \rightarrow & B \\
B & \rightarrow & T\ B_1 \mid T \\
T & \rightarrow & F\ \textbf{sub}\ T_1 \mid F \\
F & \rightarrow & (\ B\ ) \mid \textbf{text}
\end{array}
$$

The two new nonterminals, T and F, are motivated by terms and factors in expressions. Here, a "factor," generated by F, is either a parenthesized box or a text string. A "term," generated by T, is a "factor" with a sequence of subscripts, and a box generated by B is a sequence of juxtaposed "terms."

The attributes of B carry over to T and F, since the new nonterminals also denote boxes; they were introduced simply to aid parsing. Thus, both T and F have an inherited attribute ps and synthesized attributes ht and dp, with semantic actions that can be adapted from the SDT in Fig. 5.26.

The grammar is not yet ready for top-down parsing, since the productions for B and T have common prefixes. Consider T, for instance. A top-down parser cannot choose between the two productions for T by looking one symbol ahead in the input. Fortunately, we can use a form of left-factoring, discussed in Section 4.3.4, to make the grammar ready. With SDT's, the notion of com-mon prefix applies to actions as well. Both productions for T begin with the nonterminal F inheriting attribute ps from T.

The pseudocode in Fig. 5.30 for T(ps) folds in the code for F(ps). After left-factoring is applied to T -» F sub Ti | F, there is only one call to F; the pseudocode shows the result of substituting the code for F in place of this call.

The function T will be called as T(10.0) by the function for B, which we do not show. It returns a pair consisting of the height and depth of the box generated by nonterminal T; in practice, it would return a record containing the height and depth.

Function T begins by checking for a left parenthesis, in which case it must have the production F —> (B) to work with. It saves whatever the B inside the parentheses returns, but if that B is not followed by a right parenthesis, then there is a syntax error, which must be handled in a manner not shown.

Otherwise, if the current input is t e x t , then the function T uses getHt and getDp to determine the height and depth of this text.

T then decides whether the next box is a subscript and adjusts the point size, if so. We use the actions associated with the production B ->• B sub B in Fig. 5.26 for the height and depth of the larger box. Otherwise, we simply return what F would have returned: (hl,d1).

2. On-The-Fly Code Generation

The construction of long strings of code that are attribute values, as in Ex-ample 5.20, is undesirable for several reasons, including the time it could take to copy or move long strings. In common cases such as our running code-generation example, we can instead incrementally generate pieces of the code into an array or output file by executing actions in an SDT. The elements we need to make this technique work are:

```
(float, float) T(float ps) {
        float h1, h2, d1, d2; /* locals to hold heights and depths */
        /* start code for F(ps) */
        if ( current input == '(' ) {
                advance input;
                (h1, d1) = B(ps);
                if (current input != ')' ) syntax error: expected ')';
                advance input;
        }
        else if ( current input == text ) {
                let lexical value text.lexval be t;
                advance input;
                h1 = getHt(ps, t);
                d1 = getDp(ps, t);
        }
        else syntax error: expected text or '(';
        /* end code for F(ps) */
        if ( current input == sub ) {
                advance input;
                (h2, d2) = T(0.7 * ps);
                return (max(h1, h2 - 0.25 * ps), max(d1, d2 + 0.25 * ps));
        }
        return (h1, d1);
}
```

Figure 5.30: Recursive-descent typesetting of boxes

There is, for one or more nonterminals, a *main* attribute. For conve-nience, we shall assume that the main attributes are all string valued. In Example 5.20, the attributes *S.code* and *C.code* are main attributes; the other attributes are not.

The main attributes are synthesized.

3. The rules that evaluate the main attribute (s) ensure that

The main attribute is the concatenation of main attributes of non-terminals appearing in the body of the production involved, perhaps with other elements that are not main attributes, such as the string **label** or the values of labels *LI* and *L2*.

The main attributes of nonterminals appear in the rule in the same order as the nonterminals themselves appear in the production body.

As a consequence of the above conditions, the main attribute can be constructed by emitting the non-main-attribute elements of the concatenation. We can rely on the recursive calls to the functions for the nonterminals in a production body to emit the value of their main attribute incrementally.

The Type of Main Attributes

Our simplifying assumption that main attributes are of string type is really too restrictive. The true requirement is that the type of all the main attributes must have values that can be constructed by concatenation of elements. For instance, a list of objects of any type would be appropriate, as long as we represent these lists in a way that allows elements to be efficiently appended to the end of the list. Thus, if the purpose of the main attribute is to represent a sequence of intermediate-code statements, we could produce the intermediate code by writing statements to the end of an array of objects. Of course the requirements stated in Section 5.5.2 still apply to lists; for example, main attributes must be assembled from other main attributes by concatenation in order.

Example 5. 22 : We can modify the function of Fig. 5.29 to emit elements of the main translation *S.code* instead of saving them for concatenation into a return value of *S.code.* The revised function *S* appears in Fig. 5.31.

```
void S(label next) {
        label L1, L2; /* the local labels */
        if ( current input == token while ) {
                advance input;
                check '(' is next on the input, and advance;
                L1 = new();
                L2 = new();
                print("label", L1);
                C(next, L2);
                check ')' is next on the input, and advance;
                print("label", L2);
                S(L1);
        }
        else /* other statement types */
}
```

Figure 5.31: On-the-fly recursive-descent code generation for while-statements

In Fig. 5.31, *S* and *C* now have no return value, since their only synthesized attributes are produced by printing. Further, the position of the print state-ments is significant. The order in which output is printed is: first l a b e l *LI,* then the code for *C* (which is the same as the value of *Ccode* in Fig. 5.29), then label L2, and finally the code from the recursive call to S (which is the same as Scode in Fig. 5.29). Thus, the code printed by this call to S is exactly the same as the value of Scode that is returned in Fig. 5.29).

Incidentally, we can make the same change to the underlying SDT: turn the construction of a main attribute into actions that emit the elements of that attribute. In Fig. 5.32 we see the SDT of Fig. 5.28 revised to generate code on the fly.

```
S → while (   { L1 = new(); L2 = new(); C.false = S.next;
                C.true = L2; print("label", L1); }
    C )       { S1.next = L1; print("label", L2); }
    S1
```

Figure 5.32: SDT for on-the-fly code generation for while statements

Figure 5.32: SDT for on-the-fly code generation for while statements

3. L-Attributed SDD's and LL Parsing

Suppose that an L-attributed SDD is based on an LL-grammar and that we have converted it to an SDT with actions embedded in the productions, as described in Section 5.4.5. We can then perform the translation during LL parsing by extending the parser stack to hold actions and certain data items needed for attribute evaluation. Typically, the data items are copies of attributes.

In addition to records representing terminals and nonterminals, the parser stack will hold *action-records* representing actions to be executed and *synth-esize-records* to hold the synthesized attributes for nonterminals. We use the following two principles to manage attributes on the stack:

The inherited attributes of a nonterminal *A* are placed in the stack record that represents that nonterminal. The code to evaluate these attributes

will usually be represented by an action-record immediately above the stack record for *A;* in fact, the conversion of L-attributed SDD's to SDT's ensures that the action-record will be immediately above *A*.

The synthesized attributes for a nonterminal *A* are placed in a separate synthesize-record that is immediately below the record for *A* on the stack.

This strategy places records of several types on the parsing stack, trusting that these variant record types can be managed properly as subclasses of a "stack-record" class. In practice, we might

combine several records into one, but the ideas are perhaps best explained by separating data used for different purposes into different records.

Action-records contain pointers to code to be executed. Actions may also appear in synthesize-records; these actions typically place copies of the synthe-sized attribute(s) in other records further down the stack, where the value of that attribute will be needed after the synthesize-record and its attributes are popped off the stack.

Let us take a brief look at LL parsing to see the need to make temporary copies of attributes. From Section 4.4.4, a table-driven LL parser mimics a leftmost derivation. If *w* is the input that has been matched so far, then the stack holds a sequence of grammar symbols *a* such that *S wa,* where *S*

is the start symbol. When the parser expands by a production *A ->• B C,* it replaces *A* on top of the stack by *B C.*

Suppose nonterminal *C* has a n inherited attribute *C.i.* With *A B C ,* the inherited attribute *C.i* may depend not only on the inherited attributes of *A,* but on all the attributes of *B.* Thus, we may need to process *B* completely before *C.i* can be evaluated. We therefore save temporary copies of all the attributes needed to evaluate *C.i* in the action-record that evaluates *C.i.* Otherwise, when the parser replaces *A* on top of the stack by *B C,* the inherited attributes of *A* will have disappeared, along with its stack record.

Since the underlying SDD is L-attributed, we can be sure that the values of the inherited attributes of *A* are available when *A* rises to the top of the stack. The values will therefore be available in time to be copied into the action-record that evaluates the inherited attributes of *C.* Furthermore,

space for the synthesized attributes of *A* is not a problem, since the space is in the synthesize-record for *A,* which remains on the stack, below *B* and C, when the parser expands by *A -»• B C.*

As *B* is processed, we can perform actions (through a record just above *B* on the stack) that copy its inherited attributes for use by *C,* as needed, and after *B* is processed, the synthesize-record for *B* can copy its synthesized attributes for use by *C,* if needed. Likewise, synthesized attributes of *A* may need temporaries to help compute their value, and these can be copied to the synthesize-record for *A* as *B* and then *C* are processed. The principle that makes all this copying of attributes work is:

All copying takes place among the records that are created during one expansion of one nonterminal. Thus, each of these records knows how far below it on the stack each other record is, and can write values into the records below safely.

The next example illustrates the implementation of inherited attributes dur-ing LL parsing by diligently copying attribute values. Shortcuts or optimiza-tions are possible, particularly with copy rules, which simply copy the value of one attribute into another. Shortcuts are deferred until Example 5.24,. which also illustrates synthesize-records.

Example 5. 23 : This example implements the the SDT of Fig. 5.32, which generates code on the fly for the while-production. This SDT does not have synthesized attributes, except for dummy attributes that represent labels.

Figure 5.33(a) shows the situation as we are about to use the while-produc-tion to expand `5,` presumably because the lookahead symbol on the input is while . The record at the top of stack is for S, and it contains only the inherited attribute *S.next,* which we suppose has the value *x.* Since we are now parsing top-down, we show the stack top at the left, according to our usual convention.

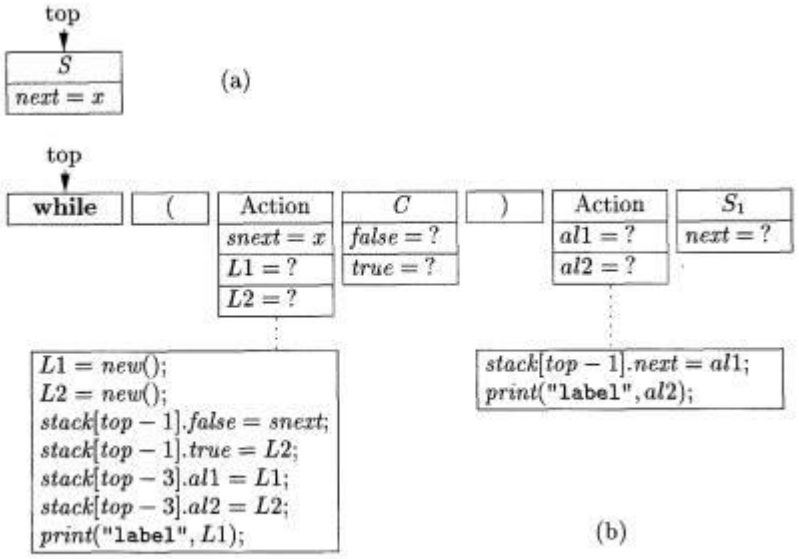


Figure 5.33: Expansion of S according to the while-statement production

Figure 5.33(b) shows the situation immediately after we have expanded S. There are action-records in front of the nonterminals *C* and Si, corresponding to the actions in the underlying SDT of Fig. 5.32. The record for *C* has room for inherited attributes *true* and *false,* while the record for Si has room for attribute *next,* as all S-records must. We show values for these fields as ?, because we do not yet know their values.

The parser next recognizes while and (on the input and pops their records off the stack. Now, the first action is at the top, and it must be executed. This action-record has a field *snext,* which holds a copy of the inherited attribute S.next. When S is popped from the stack, the value of S.next is copied into the field snext for use during the evaluation of the inherited attributes for C.

The code for the first action generates new values for LI and L2, which we shall suppose are y and z, respectively. The next step is to make z the value of C.true. The assignment stack[top — lj.true = L2 is written knowing it is only executed when this action-record is at the top of stack, so top — 1 refers to the record below it — the record for C.

The first action-record then copies LI into field all in the second action, where it will be used to evaluate Si.next. It also copies L2 into a field called all of the second action; this value is needed for that action-record to print its output properly. Finally, the first action-record prints l a b e l y to the output.

The situation after completing the first action and popping its record off

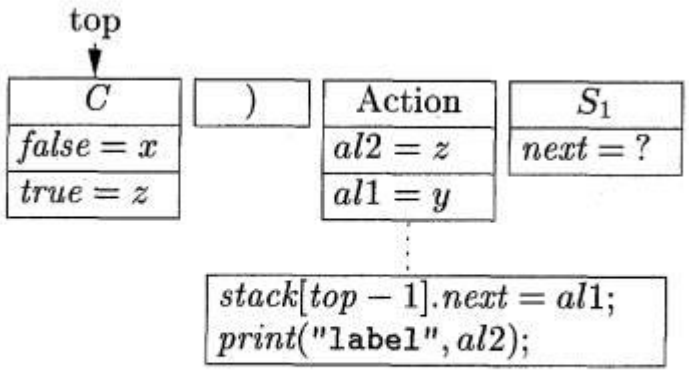


Figure 5.34: After the action above C is performed

the stack is shown in Fig. 5.34. The values of inherited attributes in the record for *C* have been filled in properly, as have the temporaries *all* and *al2* in the second action record. At this point, *C* is expanded, and we presume that the code to implement its test containing jumps to labels *x* and *z,* as appropriate, is generated. When the C-record is popped from the stack, the record for) becomes top and causes the parser to check for) on its input.

With the action above *Si* at the top of the stack, its code sets *Si.next* and emits l a b e l *z.* When that is done, the record for *Si* becomes the top of stack,

and as it is expanded, we presume it correctly generates code that implements whatever kind of statement it is and then jump to label *y.* •

E x a m p l e 5 . 24 : Now, let us consider the same while-statement, but with a translation that produces the output *S.code* as a synthesized attribute, rather than by on-the-fly generation. In order to follow the explanation, it is useful to bear in mind the following invariant or inductive hypothesis, which we assume is followed for every nonterminal:

Every nonterminal that has code associated with it leaves that code, as a string, in the synthesize-record just below it on the stack.

Assuming this statement is true, we shall handle the while-production so it maintains this statement as an invariant.

Figure 5.35(a) shows the situation just before *S* is expanded using the pro-duction for while-statements. At the top of the stack we see the record for 5; it has a field for its inherited attribute *S.next,* as in Example 5.23. Immediately below that record is the synthesize-record for this occurrence of *S.* The latter has a field for *S.code,* as all synthesize-records for *S* must have. We also show it with some other fields for local storage and actions, since the SDT for the while production in Fig. 5.28 is surely part of a larger SDT.

Our expansion of 5 is based on the SDT of Fig. 5.28, and it is shown in Fig. 5.35(b). As a shortcut, during the expansion, we assume that the inherited attribute *S.next* is assigned directly to *C.false,* rather than being placed in the first action and then copied into the record for *C.*

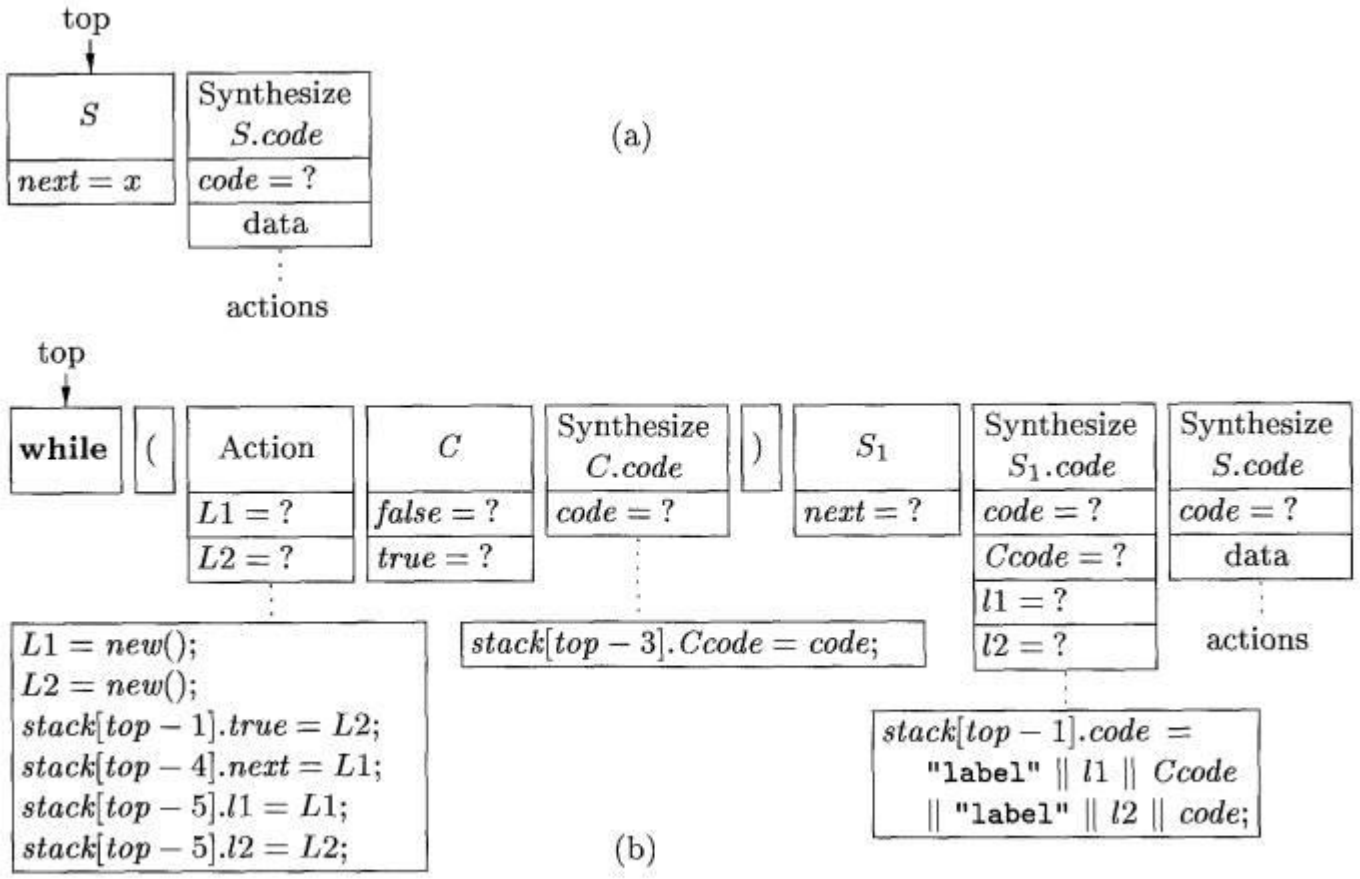


Figure 5.35: Expansion of S with synthesized attribute constructed on the stack

Let us examine what each record does when it becomes the top of stack. First, the whil e record causes the token while to be matched with the input, which it must, or else we would not have expanded *S* in this way. After a while and (are popped off the stack, the code for the action-record is executed. It generates values for LI and L2, and we take the shortcut of copying them directly to the inherited attributes that need them: *Si.next* and *C.true.* The last two steps of the action cause *LI* and L2 to be copied into the record called "Synthesize *S1.code."*

The synthesize-record for *Si* does double duty: not only will it hold the synthesized attribute *Si. code,* but it will also serve as an action-record to complete the evaluation of the attributes for the entire production *S* -» w h i l e (*C) Si.*

In particular, when it gets to the top, it will compute the synthesized attribute *S.code* and place its value in the synthesize-record for the head *S.*

When *C* becomes the top of the stack, it has both its inherited attributes computed. By the inductive hypothesis stated above, we suppose it correctly generates code to execute its condition and jump to the proper label. We also assume that the actions performed during the expansion of *C* correctly place this code in the record below, as the value of synthesized attribute *C.code.*

After *C* is popped, the synthesize-record for *C.code* becomes the top. Its code is needed in the synthesize-record for *Si.code,* because that is where we concatenate all the code elements to form *S.code.* The synthesize-record for *C.code* therefore has an action to copy *C.code* into the synthesize-record for *Si.code.* After doing so, the record for token) reaches the top of stack, and causes a check for) on the input. Assuming that test succeeds, the record for *Si* becomes the top of stack. By our inductive hypothesis, this nonterminal is expanded, and the net effect is that its code is correctly constructed and placed in the field for *code* in the synthesize-record for *Si.*

Can We Handle L-Attributed SDD's on LR Grammars?

In Section 5.4.1, we saw that every S-attributed SDD on an LR grammar can be implemented during a bottom-up parse. From Section 5.5.3 every L-attributed SDD on an LL grammar can be parsed top-down. Since LL grammars are a proper subset of the LR grammars, and the

S-attributed SDD's are a proper subset of the L-attributed SDD's, can we handle every LR grammar and L-attributed SDD bottom-up?

We cannot, as the following intuitive argument shows. Suppose we have a production A -> B C in an LR-grammar, and there is an inherited attribute B.i that depends on inherited attributes of A. When we reduce to B, we still have not seen the input that C generates, so we cannot be sure that we have a body of production A —>• B C. Thus, we cannot compute B.i yet, since we are unsure whether to use the rule associated with this production.

Perhaps we could wait until we have reduced to C, and know that we must reduce *B C* to *A*. However, even then, we do not know the inherited attributes of *A,* because even after reduction, we may not be sure of the production body that contains this *A.* We could reason that this decision, too, should be deferred, and therefore further defer the computation *of B.i.* If we keep reasoning this way, we soon realize that we cannot make any decisions until the entire input is parsed. Essentially, we have reached the strategy of "build the parse tree first and then perform the translation."

Now, all the data fields of the synthesize-record for *Si* have been filled in, so when it becomes the top of stack, the action in that record can be executed. The action causes the labels and code from *C. code* and Si . *code* to be concatenated in the proper order. The resulting string is placed in the record below; that is, in the synthesize-record for *S.* We have now correctly computed *S.code,* and when the synthesize-record for *S* becomes the top, that code is available for placement in another record further down the stack, where it will eventually be assembled into a larger string of code implementing a program element of which this *S* is a part. •

4. Bottom-Up Parsing of L-Attributed SDD's

We can do bottom-up every translation that we can do top-down. More pre-cisely, given an L-attributed SDD on an LL grammar, we can adapt the gram-mar to compute the same SDD on the new grammar during an LR parse. The "trick" has three parts:

Start with the SDT constructed as in Section 5.4.5, which places embed-ded actions before each nonterminal to compute its inherited attributes and an action at the end of the production to compute synthesized at-tributes.

Introduce into the grammar a marker nonterminal in place of each em-bedded action. Each such place gets a distinct marker, and there is one production for any marker *M,* namely *M* -> e.

Modify the action *a* if marker nonterminal *M* replaces it in some production *A* *a {a} (5,* and associate with M -> e an action *a′* that

(a) Copies, as inherited attributes of *M,* any attributes of *A* or symbols of *a* that action *a* needs.

Computes attributes in the same way as *a,* but makes those at-tributes be synthesized attributes of *M.*

This change appears illegal, since typically the action associated with production *M* —>• e will have to access attributes belonging to grammar symbols that do not appear in this production. However, we shall imple-ment the actions on the LR parsing stack, so the necessary attributes will always be available at a known number of positions down the stack.

Example 5.25 : Suppose that there is a production A -» B C in an LL grammar, and the inherited attribute B.i is computed from inherited attribute A.i by some formula B.i = f(A.i). That is, the fragment of an SDT we care about is

$$A \rightarrow \{B.i = f(A.i);\}\ B\ C$$

We introduce marker M with inherited attribute M.i and synthesized attribute M.s. The former will be a copy of A.i and the latter will be B.i. The SDT will be written

$$\begin{aligned} A &\rightarrow M\ B\ C \\ M &\rightarrow \{M.i = A.i;\ M.s = f(M.i);\} \end{aligned}$$

Notice that the rule for *M* does not have *A.i* available to it, but in fact we shall arrange that every inherited attribute for a nonterminal such as *A* appears on the stack immediately below where the reduction to *A* will later take place. Thus, when we reduce e to M, we shall find *A.i* immediately below it, from where it may be read. Also, the value of *M.s,* which is left on the stack along with *M,* is really *B.i* and properly is found right below where the reduction to *B* will later occur. •

Example 5. 2 6 : Let us turn the SDT of Fig. 5.28 into an SDT that can operate with an LR parse of the revised grammar. We introduce a marker *M* before *C* and a marker *N* before Si, so the underlying grammar becomes

$$\begin{aligned} S &\rightarrow \textbf{while}\ (\ M\ C\ )\ N\ S_1 \\ M &\rightarrow \epsilon \\ N &\rightarrow \epsilon \end{aligned}$$

Why Markers Work

Markers are nonterminals that derive only e and that appear only once among all the bodies of all productions. We shall not give a formal proof that, when a grammar is LL, marker nonterminals can be added at any position in the body, and the resulting grammar will still be LR. The intuition, however, is as follows. If a grammar is LL, then we can determine that a string *w* on the input is derived from nonterminal *A,* in a derivation that starts with production *A —> a,* by seeing only the first symbol of *w* (or the following symbol if *w* = e). Thus, if we parse *w* bottom-up, then the fact that a prefix of *w* must be reduced to *a* and then to *S* is known as soon as the beginning of *w* appears on the input. In particular, if we insert markers anywhere in *a,* the LR states will incorporate the fact that this marker has to be there, and will reduce e to the marker at the appropriate point on the input.

Before we discuss the actions that are associated with markers *M* and *N,* let us outline the "inductive hypothesis" about where attributes are stored.

Below the entire body of the while-production — that is, below **while** on the stack — will be the inherited attribute *S.next* We may not know the nonterminal or parser state associated with this stack record, but we can be sure that it will have a field, in a fixed position of the record, that

holds *S.next* before we begin to recognize what is derived from this S.

2. Inherited attributes *C.true* and *C.false* will be just below the stack record for *C.* Since the grammar is presumed to be LL, the appearance of **while** on the input assures us that the while-production is the only one that can be recognized, so we can be sure that *M* will appear immediately below *C* on the stack, and M ' s record will hold the inherited attributes of *C.*

3. Similarly, the inherited attribute *Si.next must* appear immediately below Si on the stack, so we may place that attribute in the record for *N.*

4. The synthesized attribute *C.code* will appear in the record for *C.* As always when we have a long string as an attribute value, we expect that in practice a pointer to (an object representing) the string will appear in the record, while the string itself is outside the stack.

5. Similarly, the synthesized attribute *Si.code* will appear in the record for S1 .

Let us follow the parsing process for a while-statement. Suppose that a record holding *S.next* appears on the top of the stack, and the next input is the terminal **while .** We shift this terminal onto the stack. It is then certain that the production being recognized is the while-production, so the LR parser can shift "(" and determine that its next step must be to reduce e to *M.* The stack at this time is shown in Fig. 5.36. We also show in that figure the action that is associated with the reduction to *M.* We create values for *LI* and L2, which live in fields of the M-record. Also in that record are fields for *C.true* and *C.false.* These attributes must be in the second and third fields of the record, for consistency with other stack records that might appear below *C* in other contexts and also must provide these attributes for *C.* The action completes by assigning values to *C.true* and *C.false,* one from the *L2* just generated, and the other by reaching down the stack to where we know *S.next* is found.

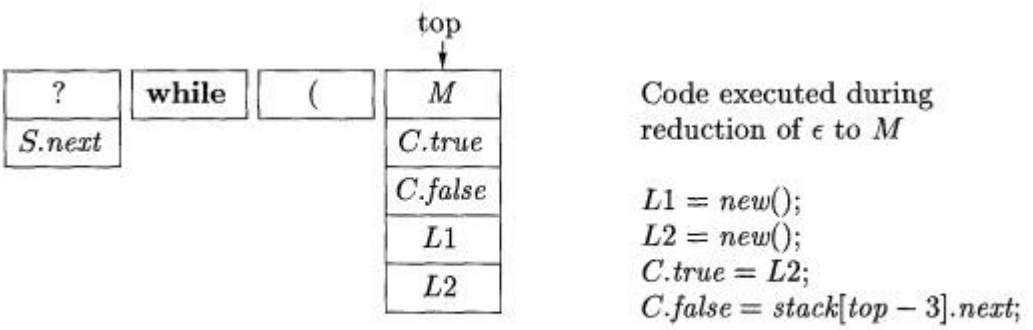


Figure 5.36: LR parsing stack after reduction of ϵ to M

We presume that the next inputs are properly reduced to *C.* The synthesized attribute *C.code* is therefore placed in the record for *C.* This change to the stack is shown in Fig. 5.37, which also incorporates the next several records that are later placed above *C* on the stack.

Continuing with the recognition of the while-statement, the parser should next find ")" on the input, which it pushes onto the stack in a record of its own. At that point, the parser, which knows it is working on a while-statement because the grammar is LL, will reduce *e to N.* The single piece of data asso-ciated with *N* is the inherited attribute *Si.next.* Note that this attribute needs to be in the record for *N* because that will be just below the record for *Si*. The code that is executed to compute the value of *Si.next* is

$$S_1.next = stack[top - 3].L1;$$

This action reaches three records below TV, which is at the top of stack when the code is executed, and retrieves the value of L I .

Next, the parser reduces some prefix of the remaining input to S, which we have consistently referred to as Si to distinguish it from the S at the head of the production. The value of Si.code is computed and appears in the stack record for Si. This step takes us to the condition that is illustrated in Fig. 5.37.

At this point, the parser will reduce everything from **while** to *Si* to *S.* The code that is executed during this reduction is:

$$\begin{aligned} &tempCode = \textbf{label} \parallel stack[top - 4].L1 \parallel stack[top - 3].code \parallel \\ &\quad \textbf{label} \parallel stack[top - 4].L2 \parallel stack[top].code; \\ &top = top - 5; \\ &stack[top].code = tempCode; \end{aligned}$$

That is, we construct the value of *S.code* in a variable *tempCode.* That code is the usual, consisting of the two labels LI and L2, the code for *C* and the code for *Si.* The stack is popped, so *S* appears

where **while** was. The value of the code for *S* is placed in the *code* field of that record, where it can be interpreted as the synthesized attribute *S.code*. Note that we do not show, in any of this discussion, the manipulation of LR states, which must also appear on the stack in the field that we have populated with grammar symbols. •

5. Exercises for Section 5.5

Exercise 5 . 5 . 1 : Implement each of your SDD's of Exercise 5.4.4 as a recursive-descent parser in the style of Section 5.5.1.

Exercise 5 . 5 . 2 : Implement each of your SDD's of Exercise 5.4.4 as a recursive-descent parser in the style of Section 5.5.2.

Exercise 5 . 5 . 3: Implement each of your SDD's of Exercise 5.4.4 with an LL parser in the style of Section 5.5.3, with code generated "on the fly."

Exercise 5 . 5 . 4: Implement each of your SDD's of Exercise 5.4.4 with an LL parser in the style of Section 5.5.3, but with code (or pointers to the code) stored on the stack.

Exercise 5.5.5 : Implement each of your SDD's of Exercise 5.4.4 with an LR parser in the style of Section 5.5.4.

Exercise 5 . 5 . 6: Implement your SDD of Exercise 5.2.4 in the style of Sec-tion 5.5.1. Would an implementation in the style of Section 5.5.2 be any differ-ent?

**********THE END**********