Chapter 5

Syntax Directed Translation

This chapter develops the theme of Section 2 3 the translation of languages guided by context free grammars The translation techniques in this chapter 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 this chapter includes an example from typesetting

We associate information with a language construct by attaching *attributes* to the grammar symbol s representing the construct as discussed in Sec tion 2 3 2 A syntax directed de nition speci es the values of attributes by associating semantic rules with the grammar productions For example an in x to post x translator might have a production and rule

PRODUCTION	SEMANTIC RULE	
$E \quad E_1 \quad T$	$E\ code \quad E_1\ code \,\|\, T\ code \,\|\, '\ '$	5 1

This production has two nonterminals E and T the subscript in E_1 distin guishes 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 speci es 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 ine cient to implement the translation directly by manipulating strings

From Section 2 3 5 a syntax directed translation scheme embeds program fragments called semantic actions within production bodies as in

$$E \quad E_1 \quad T \ \{ \text{ print } '\ ' \} \qquad 5\ 2$$

By convention semantic actions are enclosed within curly braces If curly braces occur as grammar symbols we enclose them within single quotes as in

$'\{'$ and $'\}'$ The position of a semantic action in a production body determines the order in which the action is executed In production 5 2 the action occurs at the end after all the grammar symbols in general semantic actions may occur at any position in a production body

Between the two notations syntax directed de nitions can be more readable and hence more useful for speci cations However translation schemes can be more e cient and hence more useful for implementations

The most general approach to syntax directed translation is to construct a parse tree or a syntax tree and then to compute the values of attributes at the nodes of the tree by visiting the nodes of the tree In many cases translation can be done during parsing without building an explicit tree We shall therefore study a class of syntax directed translations called L attributed translations L for left to right which encompass virtually all translations that can be performed during parsing We also study a smaller class called S attributed translations S for synthesized which can be performed easily in connection with a bottom up parse

5 1 Syntax Directed De nitions

A *syntax directed de nition* 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 elds 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

5 1 1 Inherited and Synthesized Attributes

We shall deal with two kinds of attributes for nonterminals

1 A *synthesized attribute* for a nonterminal A at a parse tree node N is de ned by a semantic rule associated with the production at N Note that the production must have A as its head A synthesized attribute at node N is de ned only in terms of attribute values at the children of N and at N itself

2 An *inherited attribute* for a nonterminal B at a parse tree node N is de ned by a semantic rule associated with the production at the parent of N Note that the production must have B as a symbol in its body An inherited attribute at node N is de ned only in terms of attribute values at N s parent N itself and N s siblings

An Alternative De nition of Inherited Attributes

No additional translations are enabled if we allow an inherited attribute B c at a node N to be de ned in terms of attribute values at the children of N as well as at N itself at its parent and at its siblings Such rules can be simulated by creating additional attributes of B say B c_1 B c_2 These are synthesized attributes that copy the needed attributes of the children of the node labeled B We then compute B c as an inherited attribute using the attributes B c_1 B c_2 in place of attributes at the children Such attributes are rarely needed in practice

While we do not allow an inherited attribute at node N to be de ned in terms of attribute values at the children of node N we do allow a synthesized attribute at node N to be de ned in terms of inherited attribute values at node N itself

Terminals can have synthesized attributes but not inherited attributes At tributes for terminals have lexical values that are supplied by the lexical ana lyzer there are no semantic rules in the SDD itself for computing the value of an attribute for a terminal

Example 5 1 The SDD in Fig 5 1 is based on our familiar grammar for arithmetic expressions with operators and It evaluates expressions termi nated by an endmarker **n** In the SDD each of the nonterminals has a single synthesized attribute called *val* We also suppose that the terminal **digit** has a synthesized attribute *lexval* which is an integer value returned by the lexical analyzer

	Production	Semantic Rules
1	L E **n**	L *val* E *val*
2	E E_1 T	E *val* E_1 *val* T *val*
3	E T	E *val* T *val*
4	T T_1 F	T *val* T_1 *val* F *val*
5	T F	T *val* F *val*
6	F E	F *val* E *val*
7	F **digit**	F *val* **digit** lexval

Figure 5 1 Syntax directed de nition of a simple desk calculator

The rule for production 1 L E **n** sets L *val* to E *val* which we shall see is the numerical value of the entire expression

Production 2 E E_1 T also has one rule which computes the *val* attribute for the head E as the sum of the values at E_1 and T At any parse tree node N labeled E the value of *val* for E is the sum of the values of *val* at the children of node N labeled E and T

Production 3 E T has a single rule that de nes the value of *val* for E to be the same as the value of *val* at the child for T Production 4 is similar to the second production its rule multiplies the values at the children instead of adding them The rules for productions 5 and 6 copy values at a child like that for the third production Production 7 gives F *val* the value of a digit that is the numerical value of the token **digit** that the lexical analyzer returned □

An SDD that involves only synthesized attributes is called *S attributed* the SDD in Fig 5 1 has this property In an S attributed SDD each rule computes an attribute for the nonterminal at the head of a production from attributes taken from the body of the production

For simplicity the examples in this section have semantic rules without side e ects In practice it is convenient to allow SDD s to have limited side e ects such as printing the result computed by a desk calculator or interacting with a symbol table Once the order of evaluation of attributes is discussed in Section 5 2 we shall allow semantic rules to compute arbitrary functions possibly involving side e ects

An S attributed SDD can be implemented naturally in conjunction with an LR parser In fact the SDD in Fig 5 1 mirrors the Yacc program of Fig 4 58 which illustrates translation during LR parsing The di erence is that in the rule for production 1 the Yacc program prints the value E *val* as a side e ect instead of de ning the attribute L *val*

An SDD without side e ects is sometimes called an *attribute grammar* The rules in an attribute grammar de ne the value of an attribute purely in terms of the values of other attributes and constants

5 1 2 Evaluating an SDD at the Nodes of a Parse Tree

To visualize the translation speci ed by an SDD it helps to work with parse trees even though a translator need not actually build a parse tree Imagine therefore that the rules of an SDD are applied by rst constructing a parse tree and then using the rules to evaluate all of the attributes at each of the nodes of the parse tree A parse tree showing the value s of its attribute s is called an *annotated parse tree*

How do we construct an annotated parse tree In what order do we evaluate attributes Before we can evaluate an attribute at a node of a parse tree we must evaluate all the attributes upon which its value depends For example if all attributes are synthesized as in Example 5 1 then we must evaluate the *val* attributes at all of the children of a node before we can evaluate the *val* attribute at the node itself

With synthesized attributes we can evaluate attributes in any bottom up order such as that of a postorder traversal of the parse tree the evaluation of S attributed de nitions is discussed in Section 5 2 3

For SDD s with both inherited and synthesized attributes there is no guar antee that there is even one order in which to evaluate attributes at nodes For instance consider nonterminals A and B with synthesized and inherited attributes $A\ s$ and $B\ i$ respectively along with the production and rules

PRODUCTION	SEMANTIC RULES
$A \quad B$	$A\ s \quad B\ i$
	$B\ i \quad A\ s \quad 1$

These rules are circular it is impossible to evaluate either $A\ s$ at a node N or $B\ i$ at the child of N without rst evaluating the other The circular dependency of $A\ s$ and $B\ i$ at some pair of nodes in a parse tree is suggested by Fig 5 2

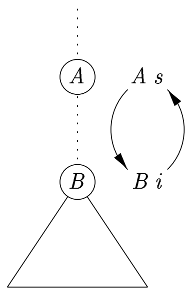


Figure 5 2 The circular dependency of $A\ s$ and $B\ i$ on one another

It is computationally di cult to determine whether or not there exist any circularities in any of the parse trees that a given SDD could have to translate [1] Fortunately there are useful subclasses of SDD s that are su cient to guarantee that an order of evaluation exists as we shall see in Section 5 2

Example 5 2 Figure 5 3 shows an annotated parse tree for the input string 3 5 4 **n** constructed using the grammar and rules of Fig 5 1 The values of *lexval* are presumed supplied by the lexical analyzer Each of the nodes for the nonterminals has attribute *val* computed in a bottom up order and we see the resulting values associated with each node For instance at the node with a child labeled after computing T *val* 3 and F *val* 5 at its rst and third children we apply the rule that says T *val* is the product of these two values or 15 □

Inherited attributes are useful when the structure of a parse tree does not match the abstract syntax of the source code The next example shows how inherited attributes can be used to overcome such a mismatch due to a grammar designed for parsing rather than translation

[1]Without going into details while the problem is decidable it cannot be solved by a polynomial time algorithm even if $\mathcal{P} \quad \mathcal{NP}$ since it has exponential time complexity

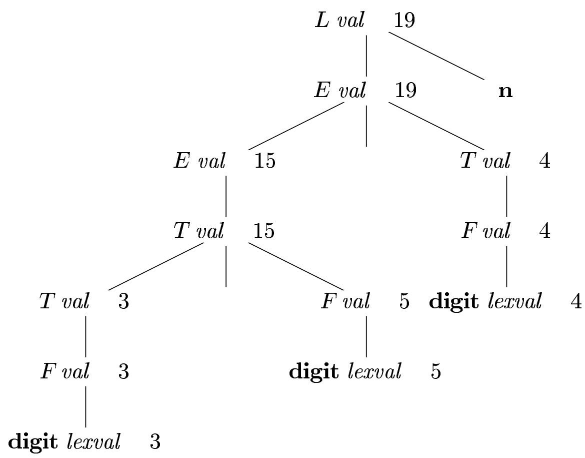


Figure 5 3 Annotated parse tree for 3 5 4 **n**

Example 5 3 The SDD in Fig 5 4 computes terms like 3 5 and 3 5 7 The top down parse of input 3 5 begins with the production $T \quad F\,T'$ Here F generates the digit 3 but the operator is generated by T' Thus the left operand 3 appears in a di erent subtree of the parse tree from An inherited attribute will therefore be used to pass the operand to the operator

The grammar in this example is an excerpt from a non left recursive version of the familiar expression grammar we used such a grammar as a running example to illustrate top down parsing in Section 4 4

	PRODUCTION	SEMANTIC RULES
1	$T \quad F\,T'$	T' *inh* F *val* T *val* T' *syn*
2	$T' \quad F\,T'_1$	T'_1 *inh* T' *inh* F *val* T' *syn* T'_1 *syn*
3	T'	T' *syn* T' *inh*
4	F **digit**	F *val* **digit** *lexval*

Figure 5 4 An SDD based on a grammar suitable for top down parsing

Each of the nonterminals T and F has a synthesized attribute *val* the terminal **digit** has a synthesized attribute *lexval* The nonterminal T' has two attributes an inherited attribute *inh* and a synthesized attribute *syn*

The semantic rules are based on the idea that the left operand of the operator is inherited More precisely the head T' of the production T' F T'_1 inherits the left operand of in the production body Given a term x y z the root of the subtree for y z inherits x Then the root of the subtree for z inherits the value of x y and so on if there are more factors in the term Once all the factors have been accumulated the result is passed back up the tree using synthesized attributes

To see how the semantic rules are used consider the annotated parse tree for 3 5 in Fig 5 5 The leftmost leaf in the parse tree labeled **digit** has attribute value *lexval* 3 where the 3 is supplied by the lexical analyzer Its parent is for production 4 F **digit** The only semantic rule associated with this production de nes F *val* **digit** *lexval* which equals 3

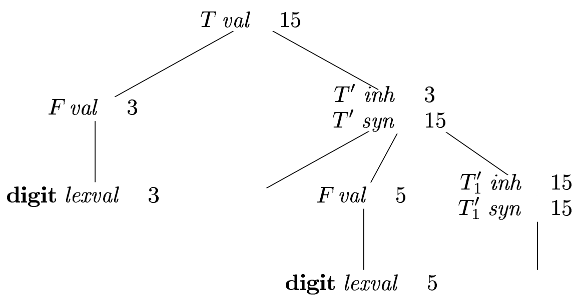


Figure 5 5 Annotated parse tree for 3 5

At the second child of the root the inherited attribute T' *inh* is de ned by the semantic rule T' *inh* F *val* associated with production 1 Thus the left operand 3 for the operator is passed from left to right across the children of the root

The production at the node for T' is T' FT'_1 We retain the subscript 1 in the annotated parse tree to distinguish between the two nodes for T' The inherited attribute T'_1 *inh* is de ned by the semantic rule T'_1 *inh* T' *inh* F *val* associated with production 2

With T' *inh* 3 and F *val* 5 we get T'_1 *inh* 15 At the lower node for T'_1 the production is T' The semantic rule T' *syn* T' *inh* de nes T'_1 *syn* 15 The *syn* attributes at the nodes for T' pass the value 15 up the tree to the node for T where T *val* 15 □

5 1 3 Exercises for Section 5 1

Exercise 5 1 1 For the SDD of Fig 5 1 give annotated parse trees for the following expressions

a 3 4 5 6 **n**

b 1 2 3 4 5 **n**

c 9 8 7 6 5 4 **n**

Exercise 5 1 2 Extend the SDD of Fig 5 4 to handle expressions as in Fig 5 1

Exercise 5 1 3 Repeat Exercise 5 1 1 using your SDD from Exercise 5 1 2

5 2 Evaluation Orders for SDD s

Dependency graphs are a useful tool for determining an evaluation order for the attribute instances in a given parse tree While an annotated parse tree shows the values of attributes a dependency graph helps us determine how those values can be computed

In this section in addition to dependency graphs we de ne two impor tant classes of SDD s the S attributed and the more general L attributed SDD s The translations speci ed by these two classes t well with the parsing methods we have studied and most translations encountered in practice can be written to conform to the requirements of at least one of these classes

5 2 1 Dependency Graphs

A *dependency graph* depicts the ow of information among the attribute in stances in a particular parse tree an edge from one attribute instance to an other means that the value of the rst is needed to compute the second Edges express constraints implied by the semantic rules In more detail

For each parse tree node say a node labeled by grammar symbol X the dependency graph has a node for each attribute associated with X

Suppose that a semantic rule associated with a production p de nes the value of synthesized attribute $A\ b$ in terms of the value of $X\ c$ the rule may de ne $A\ b$ in terms of other attributes in addition to $X\ c$ Then the dependency graph has an edge from $X\ c$ to $A\ b$ More precisely at every node N labeled A where production p is applied create an edge to attribute b at N from the attribute c at the child of N corresponding to this instance of the symbol X in the body of the production [2]

Suppose that a semantic rule associated with a production p de nes the value of inherited attribute $B\ c$ in terms of the value of $X\ a$ Then the dependency graph has an edge from $X\ a$ to $B\ c$ For each node N labeled B that corresponds to an occurrence of this B in the body of production p create an edge to attribute c at N from the attribute a at the node M

[2]Since a node can have several children labeled we again assume that subscripts distinguish among uses of the same symbol at di erent places in the production

that corresponds to this occurrence of X Note that M could be either the parent or a sibling of N

Example 5 4 Consider the following production and rule

PRODUCTION	SEMANTIC RULE
$E \quad E_1 \quad T$	$E\ val \quad E_1\ val \quad T\ val$

At every node N labeled E with children corresponding to the body of this production the synthesized attribute *val* at N is computed using the values of *val* at the two children labeled E and T Thus a portion of the dependency graph for every parse tree in which this production is used looks like Fig 5 6 As a convention we shall show the parse tree edges as dotted lines while the edges of the dependency graph are solid □

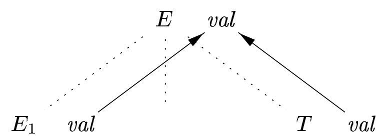


Figure 5 6 E *val* is synthesized from E_1 *val* and T *val*

Example 5 5 An example of a complete dependency graph appears in Fig 5 7 The nodes of the dependency graph represented by the numbers 1 through 9 correspond to the attributes in the annotated parse tree in Fig 5 5

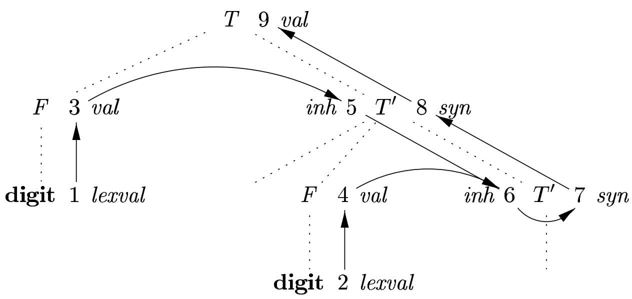


Figure 5 7 Dependency graph for the annotated parse tree of Fig 5 5

Nodes 1 and 2 represent the attribute *lexval* associated with the two leaves labeled **digit** Nodes 3 and 4 represent the attribute *val* associated with the two nodes labeled F The edges to node 3 from 1 and to node 4 from 2 result from the semantic rule that de nes $F\ val$ in terms of **digit** *lexval* In fact $F\ val$ equals **digit** *lexval* but the edge represents dependence not equality

Nodes 5 and 6 represent the inherited attribute $T'\ inh$ associated with each of the occurrences of nonterminal T' The edge to 5 from 3 is due to the rule $T'\ inh \quad F\ val$ which de nes $T'\ inh$ at the right child of the root from $F\ val$ at the left child We see edges to 6 from node 5 for $T'\ inh$ and from node 4 for $F\ val$ because these values are multiplied to evaluate the attribute *inh* at node 6

Nodes 7 and 8 represent the synthesized attribute *syn* associated with the occurrences of T' The edge to node 7 from 6 is due to the semantic rule $T'\ syn \quad T'\ inh$ associated with production 3 in Fig 5 4 The edge to node 8 from 7 is due to a semantic rule associated with production 2

Finally node 9 represents the attribute $T\ val$ The edge to 9 from 8 is due to the semantic rule $T\ val \quad T'\ syn$ associated with production 1 □

5 2 2 Ordering the Evaluation of Attributes

The dependency graph characterizes the possible orders in which we can evalu ate the attributes at the various nodes of a parse tree If the dependency graph has an edge from node M to node N then the attribute corresponding to M must be evaluated before the attribute of N Thus the only allowable orders of evaluation are those sequences of nodes $N_1\ N_2 \quad N_k$ such that if there is an edge of the dependency graph from N_i to N_j then $i \quad j$ Such an ordering embeds a directed graph into a linear order and is called a *topological sort* of the graph

If there is any cycle in the graph then there are no topological sorts that is there is no way to evaluate the SDD on this parse tree If there are no cycles however then there is always at least one topological sort To see why since there are no cycles we can surely nd a node with no edge entering For if there were no such node we could proceed from predecessor to predecessor until we came back to some node we had already seen yielding a cycle Make this node the rst in the topological order remove it from the dependency graph and repeat the process on the remaining nodes

Example 5 6 The dependency graph of Fig 5 7 has no cycles One topologi cal sort is the order in which the nodes have already been numbered 1 2 9 Notice that every edge of the graph goes from a node to a higher numbered node so this order is surely a topological sort There are other topological sorts as well such as 1 3 5 2 4 6 7 8 9 □

5 2 3 S Attributed De nitions

As mentioned earlier given an SDD it is very hard to tell whether there exist any parse trees whose dependency graphs have cycles In practice translations can be implemented using classes of SDD s that guarantee an evaluation order since they do not permit dependency graphs with cycles Moreover the two classes introduced in this section can be implemented e ciently in connection with top down or bottom up parsing

The rst class is de ned as follows

> An SDD is *S attributed* if every attribute is synthesized

Example 5 7 The SDD of Fig 5 1 is an example of an S attributed de nition Each attribute L *val* E *val* T *val* and F *val* is synthesized □

When an SDD is S attributed we can evaluate its attributes in any bottom up order of the nodes of the parse tree It is often especially simple to evaluate the attributes by performing a postorder traversal of the parse tree and evalu ating the attributes at a node N when the traversal leaves N for the last time That is we apply the function *postorder* de ned below to the root of the parse tree see also the box Preorder and Postorder Traversals in Section 2 3 4

```
postorder N {
        for  each child C of N  from the left   postorder C
        evaluate the attributes associated with node N
}
```

S attributed de nitions can be implemented during bottom up parsing since a bottom up parse corresponds to a postorder traversal Speci cally postorder corresponds exactly to the order in which an LR parser reduces a production body to its head This fact will be used in Section 5 4 2 to evaluate synthesized attributes and store them on the stack during LR parsing without creating the tree nodes explicitly

5 2 4 L Attributed De nitions

The second class of SDD s is called *L attributed de nitions* The idea behind this class is that between the attributes associated with a production body dependency graph edges can go from left to right but not from right to left hence L attributed More precisely each attribute must be either

1 Synthesized or

2 Inherited but with the rules limited as follows Suppose that there is a production A X_1X_2 X_n and that there is an inherited attribute X_i a computed by a rule associated with this production Then the rule may use only

   a Inherited attributes associated with the head A

   b Either inherited or synthesized attributes associated with the occur rences of symbols X_1 X_2 X_i $_1$ located to the left of X_i

c Inherited or synthesized attributes associated with this occurrence of X_i itself but only in such a way that there are no cycles in a dependency graph formed by the attributes of this X_i

Example 5 8 The SDD in Fig 5 4 is L attributed To see why consider the semantic rules for inherited attributes which are repeated here for convenience

PRODUCTION	SEMANTIC RULE
$T \quad F\, T'$	$T'\ inh \quad F\ val$
$T' \quad\quad F\, T_1'$	$T_1'\ inh \quad T'\ inh \quad F\ val$

The rst of these rules de nes the inherited attribute $T'\ inh$ using only $F\ val$ and F appears to the left of T' in the production body as required The second rule de nes $T_1'\ inh$ using the inherited attribute $T'\ inh$ associated with the head and $F\ val$ where F appears to the left of T_1' in the production body

In each of these cases the rules use information from above or from the left as required by the class The remaining attributes are synthesized Hence the SDD is L attributed □

Example 5 9 Any SDD containing the following production and rules cannot be L attributed

PRODUCTION	SEMANTIC RULES
$A \quad B\, C$	$A\ s \quad B\ b$
	$B\ i \quad f\ C\ c\ A\ s$

The rst rule $A\ s \quad B\ b$ is a legitimate rule in either an S attributed or L attributed SDD It de nes a synthesized attribute $A\ s$ in terms of an attribute at a child that is a symbol within the production body

The second rule de nes an inherited attribute $B\ i$ so the entire SDD cannot be S attributed Further although the rule is legal the SDD cannot be L attributed because the attribute $C\ c$ is used to help de ne $B\ i$ and C is to the right of B in the production body While attributes at siblings in a parse tree may be used in L attributed SDD s they must be to the left of the symbol whose attribute is being de ned □

5 2 5 Semantic Rules with Controlled Side E ects

In practice translations involve side e ects a desk calculator might print a result a code generator might enter the type of an identi er into a symbol table With SDD s we strike a balance between attribute grammars and translation schemes Attribute grammars have no side e ects and allow any evaluation order consistent with the dependency graph Translation schemes impose left to right evaluation and allow semantic actions to contain any program fragment translation schemes are discussed in Section 5 4

We shall control side e ects in SDD s in one of the following ways

Permit incidental side e ects that do not constrain attribute evaluation In other words permit side e ects when attribute evaluation based on any topological sort of the dependency graph produces a correct translation where correct depends on the application

Constrain the allowable evaluation orders so that the same translation is produced for any allowable order The constraints can be thought of as implicit edges added to the dependency graph

As an example of an incidental side e ect let us modify the desk calculator of Example 5 1 to print a result Instead of the rule $L\ val \quad E\ val$ which saves the result in the synthesized attribute $L\ val$ consider

	PRODUCTION	SEMANTIC RULE
1	$L \quad E$ **n**	*print* $E\ val$

Semantic rules that are executed for their side e ects such as *print* $E\ val$ will be treated as the de nitions of dummy synthesized attributes associated with the head of the production The modi ed SDD produces the same translation under any topological sort since the print statement is executed at the end after the result is computed into $E\ val$

Example 5 10 The SDD in Fig 5 8 takes a simple declaration D consisting of a basic type T followed by a list L of identi ers T can be **int** or **oat** For each identi er on the list the type is entered into the symbol table entry for the identi er We assume that entering the type for one identi er does not a ect the symbol table entry for any other identi er Thus entries can be updated in any order This SDD does not check whether an identi er is declared more than once it can be modi ed to do so

	PRODUCTION	SEMANTIC RULES
1	$D \quad T\ L$	$L\ inh \quad T\ type$
2	T **int**	$T\ type$ integer
3	T **oat**	$T\ type$ oat
4	$L \quad L_1$ **id**	$L_1\ inh \quad L\ inh$
		addType **id** *entry* $L\ inh$
5	L **id**	*addType* **id** *entry* $L\ inh$

Figure 5 8 Syntax directed de nition for simple type declarations

Nonterminal D represents a declaration which from production 1 consists of a type T followed by a list L of identi ers T has one attribute $T\ type$ which is the type in the declaration D Nonterminal L also has one attribute which we call *inh* to emphasize that it is an inherited attribute The purpose of $L\ inh$

is to pass the declared type down the list of identi ers so that it can be added to the appropriate symbol table entries

Productions 2 and 3 each evaluate the synthesized attribute T *type* giving it the appropriate value integer or oat This type is passed to the attribute L *inh* in the rule for production 1 Production 4 passes L *inh* down the parse tree That is the value L_1 *inh* is computed at a parse tree node by copying the value of L *inh* from the parent of that node the parent corresponds to the head of the production

Productions 4 and 5 also have a rule in which a function *addType* is called with two arguments

1 **id** *entry* a lexical value that points to a symbol table object and

2 L *inh* the type being assigned to every identi er on the list

We suppose that function *addType* properly installs the type L *inh* as the type of the represented identi er

A dependency graph for the input string oat **id**$_1$ **id**$_2$ **id**$_3$ appears in Fig 5 9 Numbers 1 through 10 represent the nodes of the dependency graph Nodes 1 2 and 3 represent the attribute *entry* associated with each of the leaves labeled **id** Nodes 6 8 and 10 are the dummy attributes that represent the application of the function *addType* to a type and one of these *entry* values

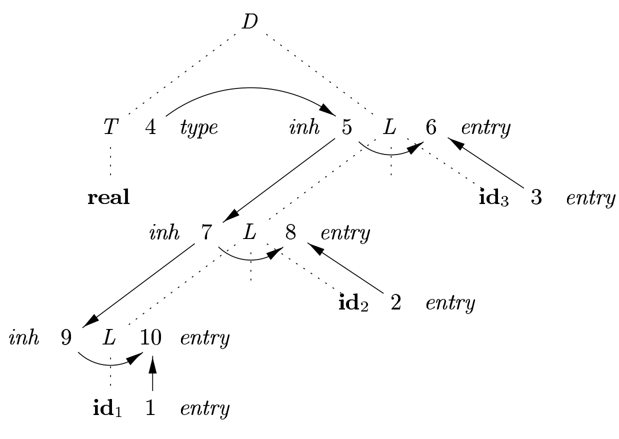


Figure 5 9 Dependency graph for a declaration oat **id**$_1$ **id**$_2$ **id**$_3$

Node 4 represents the attribute T *type* and is actually where attribute eval uation begins This type is then passed to nodes 5 7 and 9 representing L *inh* associated with each of the occurrences of the nonterminal L □

5 2 6 Exercises for Section 5 2

Exercise 5 2 1 What are all the topological sorts for the dependency graph of Fig 5 7

Exercise 5 2 2 For the SDD of Fig 5 8 give annotated parse trees for the following expressions

a `int a b c`

b `float w x y z`

Exercise 5 2 3 Suppose that we have a production $A \quad BCD$ Each of the four nonterminals A B C and D have two attributes s is a synthesized attribute and i is an inherited attribute For each of the sets of rules below tell whether i the rules are consistent with an S attributed de nition ii the rules are consistent with an L attributed de nition and iii whether the rules are consistent with any evaluation order at all

a $A\,s \quad B\,i \quad C\,s$

b $A\,s \quad B\,i \quad C\,s$ and $D\,i \quad A\,i \quad B\,s$

c $A\,s \quad B\,s \quad D\,s$

d $A\,s \quad D\,i$ $B\,i \quad A\,s \quad C\,s$ $C\,i \quad B\,s$ and $D\,i \quad B\,i \quad C\,i$

Exercise 5 2 4 This grammar generates binary numbers with a decimal point

$$\begin{array}{ll} S & L \quad L \mid L \\ L & L\,B \mid B \\ B & 0 \mid 1 \end{array}$$

Design an L attributed SDD to compute $S\,val$ the decimal number value of an input string For example the translation of string `101 101` should be the decimal number 5 625 *Hint* use an inherited attribute $L\,side$ that tells which side of the decimal point a bit is on

Exercise 5 2 5 Design an S attributed SDD for the grammar and translation described in Exercise 5 2 4

Exercise 5 2 6 Implement Algorithm 3 23 which converts a regular expres sion into a nondeterministic nite automaton by an L attributed SDD on a top down parsable grammar Assume that there is a token **char** representing any character and that **char** *lexval* is the character it represents You may also assume the existence of a function *new* that returns a new state that is a state never before returned by this function Use any convenient notation to specify the transitions of the NFA

5 3 Applications of Syntax Directed Translation

The syntax directed translation techniques in this chapter will be applied in Chapter 6 to type checking and intermediate code generation Here we consider selected examples to illustrate some representative SDD s

The main application in this section is the construction of syntax trees Since some compilers use syntax trees as an intermediate representation a common form of SDD turns its input string into a tree To complete the translation to intermediate code the compiler may then walk the syntax tree using another set of rules that are in e ect an SDD on the syntax tree rather than the parse tree Chapter 6 also discusses approaches to intermediate code generation that apply an SDD without ever constructing a tree explicitly

We consider two SDD s for constructing syntax trees for expressions The rst an S attributed de nition is suitable for use during bottom up parsing The second L attributed is suitable for use during top down parsing

The nal example of this section is an L attributed de nition that deals with basic and array types

5 3 1 Construction of Syntax Trees

As discussed in Section 2 8 2 each node in a syntax tree represents a construct the children of the node represent the meaningful components of the construct A syntax tree node representing an expression E_1 E_2 has label and two children representing the subexpressions E_1 and E_2

We shall implement the nodes of a syntax tree by objects with a suitable number of elds Each object will have an *op* eld that is the label of the node The objects will have additional elds as follows

If the node is a leaf an additional eld holds the lexical value for the leaf A constructor function *Leaf op val* creates a leaf object Alternatively if nodes are viewed as records then *Leaf* returns a pointer to a new record for a leaf

If the node is an interior node there are as many additional elds as the node has children in the syntax tree A constructor function *Node* takes two or more arguments *Node op* c_1 c_2 c_k creates an object with rst eld *op* and k additional elds for the k children c_1 c_k

Example 5 11 The S attributed de nition in Fig 5 10 constructs syntax trees for a simple expression grammar involving only the binary operators and As usual these operators are at the same precedence level and are jointly left associative All nonterminals have one synthesized attribute *node* which represents a node of the syntax tree

Every time the rst production E E_1 T is used its rule creates a node with ' ' for *op* and two children E_1 *node* and T *node* for the subexpressions The second production has a similar rule

	PRODUCTION	SEMANTIC RULES
1	$E \quad E_1 \quad T$	E *node* **new** *Node* ' ' E_1 *node* T *node*
2	$E \quad E_1 \quad T$	E *node* **new** *Node* ' ' E_1 *node* T *node*
3	$E \quad T$	E *node* T *node*
4	$T \quad E$	T *node* E *node*
5	T **id**	T *node* **new** *Leaf* **id id** *entry*
6	T **num**	T *node* **new** *Leaf* **num num** *val*

Figure 5 10 Constructing syntax trees for simple expressions

For production 3 $E \quad T$ no node is created since E *node* is the same as T *node* Similarly no node is created for production 4 $T \quad E$ The value of T *node* is the same as E *node* since parentheses are used only for grouping they in uence the structure of the parse tree and the syntax tree but once their job is done there is no further need to retain them in the syntax tree

The last two T productions have a single terminal on the right We use the constructor *Leaf* to create a suitable node which becomes the value of T *node*

Figure 5 11 shows the construction of a syntax tree for the input $a \quad 4 \quad c$ The nodes of the syntax tree are shown as records with the *op* eld rst Syntax tree edges are now shown as solid lines The underlying parse tree which need not actually be constructed is shown with dotted edges The third type of line shown dashed represents the values of E *node* and T *node* each line points to the appropriate syntax tree node

At the bottom we see leaves for a 4 and c constructed by *Leaf* We suppose that the lexical value **id** *entry* points into the symbol table and the lexical value **num** *val* is the numerical value of a constant These leaves or pointers to them become the value of T *node* at the three parse tree nodes labeled T according to rules 5 and 6 Note that by rule 3 the pointer to the leaf for a is also the value of E *node* for the leftmost E in the parse tree

Rule 2 causes us to create a node with *op* equal to the minus sign and pointers to the rst two leaves Then rule 1 produces the root node of the syntax tree by combining the node for with the third leaf

If the rules are evaluated during a postorder traversal of the parse tree or with reductions during a bottom up parse then the sequence of steps shown in Fig 5 12 ends with p_5 pointing to the root of the constructed syntax tree □

With a grammar designed for top down parsing the same syntax trees are constructed using the same sequence of steps even though the structure of the parse trees di ers signi cantly from that of syntax trees

Example 5 12 The L attributed de nition in Fig 5 13 performs the same translation as the S attributed de nition in Fig 5 10 The attributes for the grammar symbols E T **id** and **num** are as discussed in Example 5 11

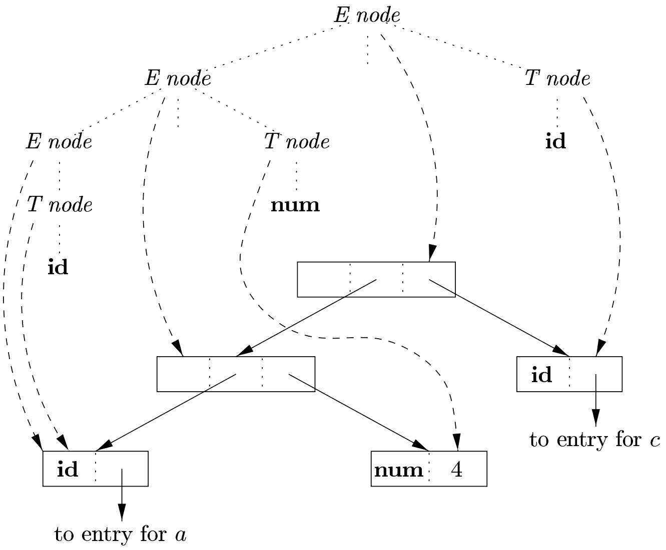


Figure 5 11 Syntax tree for a 4 c

1 p_1 **new** *Leaf* **id** *entry a*
2 p_2 **new** *Leaf* **num** 4
3 p_3 **new** *Node* ′ ′ p_1 p_2
4 p_4 **new** *Leaf* **id** *entry c*
5 p_5 **new** *Node* ′ ′ p_3 p_4

Figure 5 12 Steps in the construction of the syntax tree for a 4 c

The rules for building syntax trees in this example are similar to the rules for the desk calculator in Example 5 3 In the desk calculator example a term x y was evaluated by passing x as an inherited attribute since x and y appeared in di erent portions of the parse tree Here the idea is to build a syntax tree for x y by passing x as an inherited attribute since x and y appear in di erent subtrees Nonterminal E' is the counterpart of nonterminal T' in Example 5 3 Compare the dependency graph for a 4 c in Fig 5 14 with that for 3 5 in Fig 5 7

Nonterminal E' has an inherited attribute *inh* and a synthesized attribute *syn* Attribute E' *inh* represents the partial syntax tree constructed so far Speci cally it represents the root of the tree for the pre x of the input string that is to the left of the subtree for E' At node 5 in the dependency graph in Fig 5 14 E' *inh* denotes the root of the partial syntax tree for the identi er a that is the leaf for a At node 6 E' *inh* denotes the root for the partial syntax

	PRODUCTION	SEMANTIC RULES
1	$E \quad T\,E'$	$E\ node \quad E'\ syn$ $E'\ inh \quad T\ node$
2	$E' \qquad T\,E'_1$	$E'_1\ inh$ **new** $Node\ ' \quad '\ E'\ inh\ T\ node$ $E'\ syn \quad E'_1\ syn$
3	$E' \qquad T\,E'_1$	$E'_1\ inh$ **new** $Node\ ' \quad '\ E'\ inh\ T\ node$ $E'\ syn \quad E'_1\ syn$
4	E'	$E'\ syn \quad E'\ inh$
5	$T \quad E$	$T\ node \quad E\ node$
6	T **id**	$T\ node$ **new** $Leaf$ **id id** $entry$
7	T **num**	$T\ node$ **new** $Leaf$ **num num** val

Figure 5 13 Constructing syntax trees during top down parsing

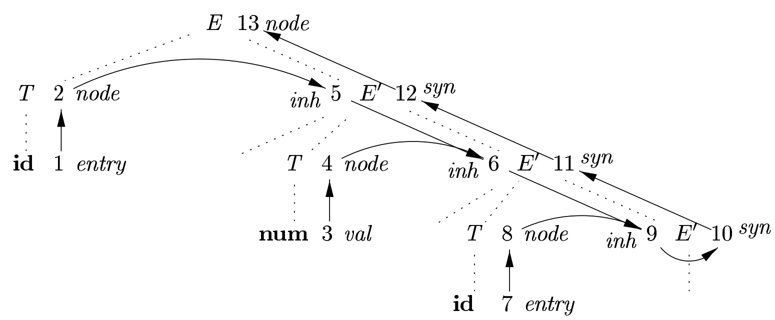


Figure 5 14 Dependency graph for $a \quad 4 \quad c$ with the SDD of Fig 5 13

tree for the input $a \quad 4$ At node 9 $E'\ inh$ denotes the syntax tree for $a \quad 4 \quad c$

Since there is no more input at node 9 $E'\ inh$ points to the root of the entire syntax tree The *syn* attributes pass this value back up the parse tree until it becomes the value of $E\ node$ Speci cally the attribute value at node 10 is de ned by the rule $E'\ syn \quad E'\ inh$ associated with the production E' The attribute value at node 11 is de ned by the rule $E'\ syn \quad E'_1\ syn$ associated with production 2 in Fig 5 13 Similar rules de ne the attribute values at nodes 12 and 13 □

5 3 2 The Structure of a Type

Inherited attributes are useful when the structure of the parse tree di ers from the abstract syntax of the input attributes can then be used to carry informa tion from one part of the parse tree to another The next example shows how a mismatch in structure can be due to the design of the language and not due to constraints imposed by the parsing method

Example 5 13 In C the type **int** 2 3 can be read as array of 2 arrays of 3 integers The corresponding type expression *array* 2 *array* 3 *integer* is represented by the tree in Fig 5 15 The operator *array* takes two parameters a number and a type If types are represented by trees then this operator returns a tree node labeled *array* with two children for a number and a type

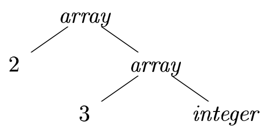


Figure 5 15 Type expression for **int** 2 3

With the SDD in Fig 5 16 nonterminal T generates either a basic type or an array type Nonterminal B generates one of the basic types **int** and **oat** T generates a basic type when T derives $B\,C$ and C derives Otherwise C generates array components consisting of a sequence of integers each integer surrounded by brackets

PRODUCTION	SEMANTIC RULES
T $B\ C$	$T\,t$ $C\,t$
	$C\,b$ $B\,t$
B **int**	$B\,t$ *integer*
B **oat**	$B\,t$ *oat*
C **num** C_1	$C\,t$ *array* **num** *val* $C_1\,t$
	$C_1\,b$ $C\,b$
C	$C\,t$ $C\,b$

Figure 5 16 T generates either a basic type or an array type

The nonterminals B and T have a synthesized attribute t representing a type The nonterminal C has two attributes an inherited attribute b and a synthesized attribute t The inherited b attributes pass a basic type down the tree and the synthesized t attributes accumulate the result

An annotated parse tree for the input string **int** 2 3 is shown in Fig 5 17 The corresponding type expression in Fig 5 15 is constructed by passing the type *integer* from B down the chain of C s through the inherited attributes b The array type is synthesized up the chain of C s through the attributes t

In more detail at the root for T $B\,C$ nonterminal C inherits the type from B using the inherited attribute $C\,b$ At the rightmost node for C the production is C so $C\ t$ equals $C\ b$ The semantic rules for the production C **num** C_1 form $C\ t$ by applying the operator *array* to the operands **num** *val* and $C_1\ t$ □

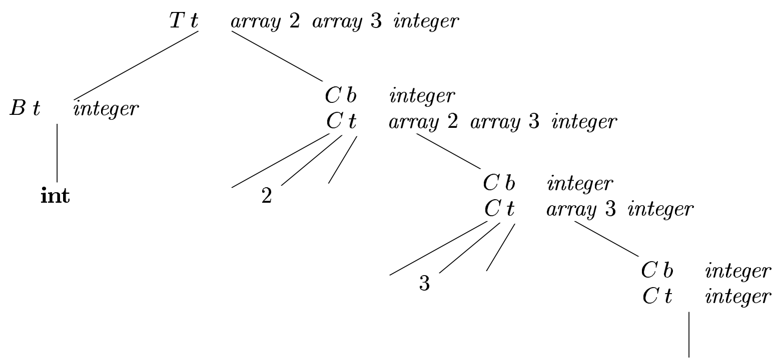


Figure 5 17 Syntax directed translation of array types

5 3 3 Exercises for Section 5 3

Exercise 5 3 1 Below is a grammar for expressions involving operator and integer or oating point operands Floating point numbers are distinguished by having a decimal point

$$\begin{array}{lll} E & E & T \mid T \\ T & \textbf{num} & \textbf{num} \mid \textbf{num} \end{array}$$

a Give an SDD to determine the type of each term T and expression E

b Extend your SDD of a to translate expressions into post x notation Use the unary operator **intToFloat** to turn an integer into an equivalent oat

Exercise 5 3 2 Give an SDD to translate in x expressions with and into equivalent expressions without redundant parentheses For example since both operators associate from the left and takes precedence over a b c d translates into a b c d

Exercise 5 3 3 Give an SDD to di erentiate expressions such as x 3 x x x involving the operators and the variable x and constants Assume that no simpli cation occurs so that for example 3 x will be translated into 3 1 0 x

5 4 Syntax Directed Translation Schemes

Syntax directed translation schemes are a complementary notation to syntax directed de nitions All of the applications of syntax directed de nitions in Section 5 3 can be implemented using syntax directed translation schemes

From Section 2 3 5 a *syntax directed translation scheme* SDT is a context free grammar with program fragments embedded within production bodies The program fragments are called *semantic actions* and can appear at any position within a production body By convention we place curly braces around actions if braces are needed as grammar symbols then we quote them

Any SDT can be implemented by rst building a parse tree and then per forming the actions in a left to right depth rst order that is during a preorder traversal An example appears in Section 5 4 3

Typically SDT s are implemented during parsing without building a parse tree In this section we focus on the use of SDT s to implement two important classes of SDD s

1 The underlying grammar is LR parsable and the SDD is S attributed

2 The underlying grammar is LL parsable and the SDD is L attributed

We shall see how in both these cases the semantic rules in an SDD can be converted into an SDT with actions that are executed at the right time During parsing an action in a production body is executed as soon as all the grammar symbols to the left of the action have been matched

SDT s that can be implemented during parsing can be characterized by in troducing distinct *marker nonterminals* in place of each embedded action each marker M has only one production M If the grammar with marker non terminals can be parsed by a given method then the SDT can be implemented during parsing

5 4 1 Post x Translation Schemes

By far the simplest SDD implementation occurs when we can parse the grammar bottom up and the SDD is S attributed In that case we can construct an SDT in which each action is placed at the end of the production and is executed along with the reduction of the body to the head of that production SDT s with all actions at the right ends of the production bodies are called *post x SDT s*

Example 5 14 The post x SDT in Fig 5 18 implements the desk calculator SDD of Fig 5 1 with one change the action for the rst production prints a value The remaining actions are exact counterparts of the semantic rules Since the underlying grammar is LR and the SDD is S attributed these actions can be correctly performed along with the reduction steps of the parser □

L	E **n**	{ print E *val* }
E	E_1 T	{ E *val* E_1 *val* T *val* }
E	T	{ E *val* T *val* }
T	T_1 F	{ T *val* T_1 *val* F *val* }
T	F	{ T *val* F *val* }
F	E	{ F *val* E *val* }
F	**digit**	{ F *val* **digit** *lexval* }

Figure 5 18 Post x SDT implementing the desk calculator

5 4 2 Parser Stack Implementation of Post x SDT s

Post x SDT s can be implemented during LR parsing by executing the actions when reductions occur The attribute s of each grammar symbol can be put on the stack in a place where they can be found during the reduction The best plan is to place the attributes along with the grammar symbols or the LR states that represent these symbols in records on the stack itself

In Fig 5 19 the parser stack contains records with a eld for a grammar symbol or parser state and below it a eld for an attribute The three grammar symbols $X\,Y\,Z$ are on top of the stack perhaps they are about to be reduced according to a production like A $X\,Y\,Z$ Here we show X x as the one attribute of X and so on In general we can allow for more attributes either by making the records large enough or by putting pointers to records on the stack With small attributes it may be simpler to make the records large enough even if some elds go unused some of the time However if one or more attributes are of unbounded size say they are character strings then it would be better to put a pointer to the attribute s value in the stack record and store the actual value in some larger shared storage area that is not part of the stack

	X	Y	Z	State grammar symbol
	X x	Y y	Z z	Synthesized attribute s
			top	

Figure 5 19 Parser stack with a eld for synthesized attributes

If the attributes are all synthesized and the actions occur at the ends of the productions then we can compute the attributes for the head when we reduce the body to the head If we reduce by a production such as A $X\,Y\,Z$ then we have all the attributes of X Y and Z available at known positions on the stack as in Fig 5 19 After the action A and its attributes are at the top of the stack in the position of the record for X

Example 5 15 Let us rewrite the actions of the desk calculator SDT of Ex ample 5 14 so that they manipulate the parser stack explicitly Such stack manipulation is usually done automatically by the parser

PRODUCTION	ACTIONS
L E **n**	{ print *stack top* 1 *val* *top* *top* 1 }
E E_1 T	{ *stack top* 2 *val* *stack top* 2 *val* *stack top val* *top* *top* 2 }
E T	
T T_1 F	{ *stack top* 2 *val* *stack top* 2 *val* *stack top val* *top* *top* 2 }
T F	
F E	{ *stack top* 2 *val* *stack top* 1 *val* *top* *top* 2 }
F **digit**	

Figure 5 20 Implementing the desk calculator on a bottom up parsing stack

Suppose that the stack is kept in an array of records called *stack* with *top* a cursor to the top of the stack Thus *stack top* refers to the top record on the stack *stack top* 1 to the record below that and so on Also we assume that each record has a eld called *val* which holds the attribute of whatever grammar symbol is represented in that record Thus we may refer to the attribute *E val* that appears at the third position on the stack as *stack top* 2 *val* The entire SDT is shown in Fig 5 20

For instance in the second production E E_1 T we go two positions below the top to get the value of E_1 and we nd the value of T at the top The resulting sum is placed where the head E will appear after the reduction that is two positions below the current top The reason is that after the reduction the three topmost stack symbols are replaced by one After computing *E val* we pop two symbols o the top of the stack so the record where we placed *E val* will now be at the top of the stack

In the third production E T no action is necessary because the length of the stack does not change and the value of *T val* at the stack top will simply become the value of *E val* The same observation applies to the productions T F and F **digit** Production F E is slightly di erent Although the value does not change two positions are removed from the stack during the reduction so the value has to move to the position after the reduction

Note that we have omitted the steps that manipulate the rst eld of the stack records the eld that gives the LR state or otherwise represents the grammar symbol If we are performing an LR parse the parsing table tells us what the new state is every time we reduce see Algorithm 4 44 Thus we may simply place that state in the record for the new top of stack □

5 4 3 SDT s With Actions Inside Productions

An action may be placed at any position within the body of a production It is performed immediately after all symbols to its left are processed Thus if we have a production B X $\{a\}$ Y the action a is done after we have recognized X if X is a terminal or all the terminals derived from X if X is a nonterminal More precisely

If the parse is bottom up then we perform action a as soon as this oc currence of X appears on the top of the parsing stack

If the parse is top down we perform a just before we attempt to expand this occurrence of Y if Y a nonterminal or check for Y on the input if Y is a terminal

SDT s that can be implemented during parsing include post x SDT s and a class of SDT s considered in Section 5 5 that implements L attributed de ni tions Not all SDT s can be implemented during parsing as we shall see in the next example

Example 5 16 As an extreme example of a problematic SDT suppose that we turn our desk calculator running example into an SDT that prints the pre x form of an expression rather than evaluating the expression The productions and actions are shown in Fig 5 21

1	L	E **n**
2	E	{ print ' ' } E_1 T
3	E	T
4	T	{ print ' ' } T_1 F
5	T	F
6	F	E
7	F	**digit** { print **digit** *lexval* }

Figure 5 21 Problematic SDT for in x to pre x translation during parsing

Unfortunately it is impossible to implement this SDT during either top down or bottom up parsing because the parser would have to perform critical actions like printing instances of or long before it knows whether these symbols will appear in its input

Using marker nonterminals M_2 and M_4 for the actions in productions 2 and 4 respectively on input that is a digit a shift reduce parser see Sec tion 4 5 3 has con icts between reducing by M_2 reducing by M_4 and shifting the digit □

Any SDT can be implemented as follows

1 Ignoring the actions parse the input and produce a parse tree as a result

2 Then examine each interior node N say one for production A Add additional children to N for the actions in so the children of N from left to right have exactly the symbols and actions of

3 Perform a preorder traversal see Section 2 3 4 of the tree and as soon as a node labeled by an action is visited perform that action

For instance Fig 5 22 shows the parse tree for expression 3 5 4 with ac tions inserted If we visit the nodes in preorder we get the pre x form of the expression 3 5 4

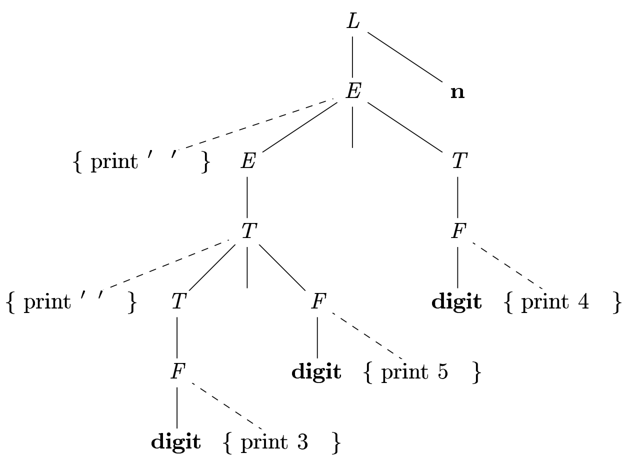


Figure 5 22 Parse tree with actions embedded

5 4 4 Eliminating Left Recursion From SDT s

Since no grammar with left recursion can be parsed deterministically top down we examined left recursion elimination in Section 4 3 3 When the grammar is part of an SDT we also need to worry about how the actions are handled

First consider the simple case in which the only thing we care about is the order in which the actions in an SDT are performed For example if each action simply prints a string we care only about the order in which the strings are printed In this case the following principle can guide us

> When transforming the grammar treat the actions as if they were termi nal symbols

This principle is based on the idea that the grammar transformation preserves the order of the terminals in the generated string The actions are therefore executed in the same order in any left to right parse top down or bottom up

The trick for eliminating left recursion is to take two productions

$$A \quad A \quad |$$

that generate strings consisting of a and any number of s and replace them by productions that generate the same strings using a new nonterminal R for remainder of the rst production

$$\begin{array}{ll} A & R \\ R & R\,| \end{array}$$

If does not begin with A then A no longer has a left recursive production In regular de nition terms with both sets of productions A is de ned by See Section 4 3 3 for the handling of situations where A has more recursive or nonrecursive productions

Example 5 17 Consider the following E productions from an SDT for trans lating in x expressions into post x notation

$$\begin{array}{lll} E & E_1 \quad T & \{ \text{ print } ' \quad ' \ \} \\ E & T & \end{array}$$

If we apply the standard transformation to E the remainder of the left recursive production is

$$T \ \{ \text{ print } ' \quad ' \ \}$$

and the body of the other production is T If we introduce R for the remain der of E we get the set of productions

$$\begin{array}{ll} E & T\,R \\ R & \quad T \ \{ \text{ print } ' \quad ' \ \} \ R \\ R & \end{array}$$

□

When the actions of an SDD compute attributes rather than merely printing output we must be more careful about how we eliminate left recursion from a grammar However if the SDD is S attributed then we can always construct an SDT by placing attribute computing actions at appropriate positions in the new productions

We shall give a general schema for the case of a single recursive production a single nonrecursive production and a single attribute of the left recursive nonterminal the generalization to many productions of each type is not hard but is notationally cumbersome Suppose that the two productions are

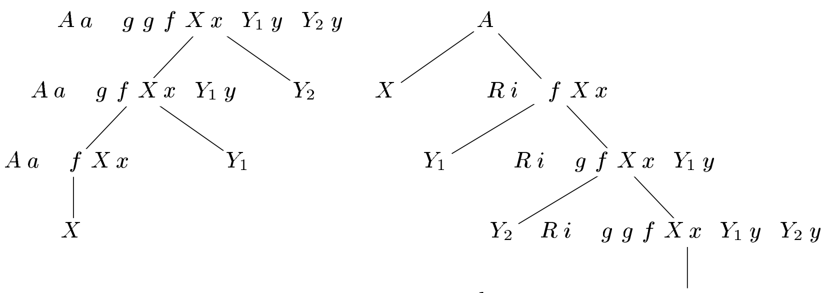
$$A \quad A_1\, Y\ \{A\, a \quad g\ A_1\, a\ Y\, y\ \}$$
$$A \quad X\ \{A\, a \quad f\ X\, x\ \}$$

Here $A\, a$ is the synthesized attribute of left recursive nonterminal A and X and Y are single grammar symbols with synthesized attributes $X\, x$ and $Y\, y$ respectively These could represent a string of several grammar symbols each with its own attribute s since the schema has an arbitrary function g computing $A\, a$ in the recursive production and an arbitrary function f computing $A\, a$ in the second production In each case f and g take as arguments whatever attributes they are allowed to access if the SDD is S attributed

We want to turn the underlying grammar into

$$A \quad X\, R$$
$$R \quad Y\, R \mid$$

Figure 5 23 suggests what the SDT on the new grammar must do In a we see the e ect of the post x SDT on the original grammar We apply f once corresponding to the use of production A X and then apply g as many times as we use the production A $A\, Y$ Since R generates a remainder of Y s its translation depends on the string to its left a string of the form XYY Y Each use of the production R $Y R$ results in an application of g For R we use an inherited attribute $R\, i$ to accumulate the result of successively applying g starting with the value of $A\, a$



a b

Figure 5 23 Eliminating left recursion from a post x SDT

In addition R has a synthesized attribute $R\, s$ not shown in Fig 5 23 This attribute is rst computed when R ends its generation of Y symbols as signaled by the use of production R $R\, s$ is then copied up the tree so it can become the value of $A\, a$ for the entire expression XYY Y The case where A generates XYY is shown in Fig 5 23 and we see that the value of $A\, a$ at the root of a has two uses of g So does $R\, i$ at the bottom of tree b and it is this value of $R\, s$ that gets copied up that tree

To accomplish this translation we use the following SDT

$$\begin{array}{lll} A & & X\ \{R\,i \quad f\ X\,x\ \}\ R\ \{A\,a \quad R\,s\} \\ R & & Y\ \{R_1\,i \quad g\ R\,i\ Y\,y\ \}\ R_1\ \{R\,s \quad R_1\,s\} \\ R & & \quad \{R\,s \quad R\,i\} \end{array}$$

Notice that the inherited attribute $R\,i$ is evaluated immediately before a use of R in the body while the synthesized attributes $A\,a$ and $R\,s$ are evaluated at the ends of the productions Thus whatever values are needed to compute these attributes will be available from what has been computed to the left

5 4 5 SDT s for L Attributed De nitions

In Section 5 4 1 we converted S attributed SDD s into post x SDT s with actions at the right ends of productions As long as the underlying grammar is LR post x SDT s can be parsed and translated bottom up

Now we consider the more general case of an L attributed SDD We shall assume that the underlying grammar can be parsed top down for if not it is frequently impossible to perform the translation in connection with either an LL or an LR parser With any grammar the technique below can be imple mented by attaching actions to a parse tree and executing them during preorder traversal of the tree

The rules for turning an L attributed SDD into an SDT are as follows

1 Embed the action that computes the inherited attributes for a nonterminal A immediately before that occurrence of A in the body of the production If several inherited attributes for A depend on one another in an acyclic fashion order the evaluation of attributes so that those needed rst are computed rst

2 Place the actions that compute a synthesized attribute for the head of a production at the end of the body of that production

We shall illustrate these principles with two extended examples The rst involves typesetting It illustrates how the techniques of compiling can be used in language processing for applications other than what we normally think of as programming languages The second example is about the generation of intermediate code for a typical programming language construct a form of while statement

Example 5 18 This example is motivated by languages for typesetting math ematical formulas `Eqn` is an early example of such a language ideas from `Eqn` are still found in the TEX typesetting system which was used to produce this book

We shall concentrate on only the capability to de ne subscripts subscripts of subscripts and so on ignoring superscripts built up fractions and all other mathematical features In the `Eqn` language one writes `a sub i sub j` to set the expression a_{i_j} A simple grammar for *boxes* elements of text bounded by a rectangle is

$$B \quad B_1\ B_2 \mid B_1 \ \textbf{sub}\ B_2 \mid \quad B_1 \quad \mid \textbf{text}$$

Corresponding to these four productions a box can be either

1 Two boxes juxtaposed with the rst B_1 to the left of the other B_2

2 A box and a subscript box The second box appears in a smaller size lower and to the right of the rst box

3 A parenthesized box for grouping of boxes and subscripts `Eqn` and TEX both use curly braces for grouping but we shall use ordinary round paren theses to avoid confusion with the braces that surround actions in SDT s

4 A text string that is any string of characters

This grammar is ambiguous but we can still use it to parse bottom up if we make subscripting and juxtaposition right associative with **sub** taking prece dence over juxtaposition

Expressions will be typeset by constructing larger boxes out of smaller ones In Fig 5 24 the boxes for E_1 and *height* are about to be juxtaposed to form the box for E_1 *height* The left box for E_1 is itself constructed from the box for E and the subscript 1 The subscript 1 is handled by shrinking its box by about 30 lowering it and placing it after the box for E Although we shall treat *height* as a text string the rectangles within its box show how it can be constructed from boxes for the individual letters

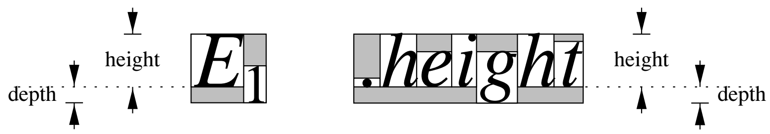


Figure 5 24 Constructing larger boxes from smaller ones

In this example we concentrate on the vertical geometry of boxes only The horizontal geometry the widths of boxes is also interesting especially when di erent characters have di erent widths It may not be readily apparent but each of the distinct characters in Fig 5 24 has a di erent width

The values associated with the vertical geometry of boxes are as follows

a The *point size* is used to set text within a box We shall assume that characters not in subscripts are set in 10 point type the size of type in this book Further we assume that if a box has point size p then its subscript box has the smaller point size 0 $7p$ Inherited attribute B ps will represent the point size of block B This attribute must be inherited because the context determines by how much a given box needs to be shrunk due to the number of levels of subscripting b Each box has a *baseline* which is a vertical position that corresponds to the bottoms of lines of text not counting any letters like g that extend below the normal baseline In Fig 5 24 the dotted line represents the baseline for the boxes E *height* and the entire expression The baseline for the box containing the subscript 1 is adjusted to lower the subscript

c A box has a *height* which is the distance from the top of the box to the baseline Synthesized attribute $B\ ht$ gives the height of box B

d A box has a *depth* which is the distance from the baseline to the bottom of the box Synthesized attribute $B\ dp$ gives the depth of box B

The SDD in Fig 5 25 gives rules for computing point sizes heights and depths Production 1 is used to assign $B\ ps$ the initial value 10

	Production	Semantic Rules
1	$S \quad B$	$B\ ps \quad 10$
2	$B \quad B_1\ B_2$	$B_1\ ps \quad B\ ps$ $B_2\ ps \quad B\ ps$ $B\ ht \quad \max\ B_1\ ht\ B_2\ ht$ $B\ dp \quad \max\ B_1\ dp\ B_2\ dp$
3	$B \quad B_1$ **sub** B_2	$B_1\ ps \quad B\ ps$ $B_2\ ps \quad 0\ 7 \quad B\ ps$ $B\ ht \quad \max\ B_1\ ht\ B_2\ ht \quad 0\ 25 \quad B\ ps$ $B\ dp \quad \max\ B_1\ dp\ B_2\ dp \quad 0\ 25 \quad B\ ps$
4	$B \quad B_1$	$B_1\ ps \quad B\ ps$ $B\ ht \quad B_1\ ht$ $B\ dp \quad B_1\ dp$
5	B **text**	$B\ ht \quad getHt\ B\ ps$ **text** *lexval* $B\ dp \quad getDp\ B\ ps$ **text** *lexval*

Figure 5 25 SDD for typesetting boxes

Production 2 handles juxtaposition Point sizes are copied down the parse tree that is two sub boxes of a box inherit the same point size from the larger box Heights and depths are computed up the tree by taking the maximum That is the height of the larger box is the maximum of the heights of its two components and similarly for the depth

Production 3 handles subscripting and is the most subtle In this greatly simpli ed example we assume that the point size of a subscripted box is 70 of the point size of its parent Reality is much more complex since subscripts cannot shrink inde nitely in practice after a few levels the sizes of subscripts shrink hardly at all Further we assume that the baseline of a subscript box drops by 25 of the parent s point size again reality is more complex

Production 4 copies attributes appropriately when parentheses are used Fi nally production 5 handles the leaves that represent text boxes In this matter too the true situation is complicated so we merely show two unspeci ed func tions *getHt* and *getDp* that examine tables created with each font to determine the maximum height and maximum depth of any characters in the text string The string itself is presumed to be provided as the attribute *lexval* of terminal **text**

Our last task is to turn this SDD into an SDT following the rules for an L attributed SDD which Fig 5 25 is The appropriate SDT is shown in Fig 5 26 For readability since production bodies become long we split them across lines and line up the actions Production bodies therefore consist of the contents of all lines up to the head of the next production □

	Production		Actions	
1	S		{ B *ps*	10 }
		B		
2	B		{B_1 *ps*	B *ps* }
		B_1	{B_2 *ps*	B *ps* }
		B_2	{ B *ht*	max B_1 *ht* B_2 *ht*
			B *dp*	max B_1 *dp* B_2 *dp* }
3	B		{B_1 *ps*	B *ps* }
		B_1 **sub**	{B_2 *ps*	0 7 B *ps* }
		B_2	{ B *ht*	max B_1 *ht* B_2 *ht* 0 25 B *ps*
			B *dp*	max B_1 *dp* B_2 *dp* 0 25 B *ps* }
4	B		{B_1 *ps*	B *ps* }
		B_1	{ B *ht*	B_1 *ht*
			B *dp*	B_1 *dp* }
5	B	**text**	{ B *ht*	*getHt* B *ps* **text** *lexval*
			B *dp*	*getDp* B *ps* **text** *lexval* }

Figure 5 26 SDT for typesetting boxes

Our next example concentrates on a simple while statement and the gener ation of intermediate code for this type of statement Intermediate code will be treated as a string valued attribute Later we shall explore techniques that involve the writing of pieces of a string valued attribute as we parse thus avoid ing the copying of long strings to build even longer strings The technique was introduced in Example 5 17 where we generated the post x form of an in x expression on the y rather than computing it as an attribute However in our rst formulation we create a string valued attribute by concatenation

Example 5 19 For this example we only need one production

$$S \quad \textbf{while} \quad C \quad S_1$$

Here S is the nonterminal that generates all kinds of statements presumably including if statements assignment statements and others In this example C stands for a conditional expression a boolean expression that evaluates to true or false

In this ow of control example the only things we ever generate are labels All the other intermediate code instructions are assumed to be generated by parts of the SDT that are not shown Speci cally we generate explicit instruc tions of the form **label** L where L is an identi er to indicate that L is the label of the instruction that follows We assume that the intermediate code is like that introduced in Section 2 8 4

The meaning of our while statement is that the conditional C is evaluated If it is true control goes to the beginning of the code for S_1 If false then control goes to the code that follows the while statement s code The code for S_1 must be designed to jump to the beginning of the code for the while statement when nished the jump to the beginning of the code that evaluates C is not shown in Fig 5 27

We use the following attributes to generate the proper intermediate code

1 The inherited attribute S *next* labels the beginning of the code that must be executed after S is nished

2 The synthesized attribute S *code* is the sequence of intermediate code steps that implements a statement S and ends with a jump to S *next*

3 The inherited attribute C *true* labels the beginning of the code that must be executed if C is true

4 The inherited attribute C *false* labels the beginning of the code that must be executed if C is false

5 The synthesized attribute C *code* is the sequence of intermediate code steps that implements the condition C and jumps either to C *true* or to C *false* depending on whether C is true or false

The SDD that computes these attributes for the while statement is shown in Fig 5 27 A number of points merit explanation

The function *new* generates new labels

The variables $L1$ and $L2$ hold labels that we need in the code $L1$ is the beginning of the code for the while statement and we need to arrange

S	**while** C S_1	$L1$ *new*
		$L2$ *new*
		S_1 *next* $L1$
		C *false* S *next*
		C *true* $L2$
		S *code* **label** $\|\|$ $L1$ $\|\|$ C *code* $\|\|$ **label** $\|\|$ $L2$ $\|\|$ S_1 *code*

Figure 5 27 SDD for while statements

that S_1 jumps there after it nishes That is why we set S_1 *next* to $L1$ $L2$ is the beginning of the code for S_1 and it becomes the value of C *true* because we branch there when C is true

Notice that C *false* is set to S *next* because when the condition is false we execute whatever code must follow the code for S

We use $||$ as the symbol for concatenation of intermediate code fragments The value of S *code* thus begins with the label $L1$ then the code for condition C another label $L2$ and the code for S_1

This SDD is L attributed When we convert it into an SDT the only re maining issue is how to handle the labels $L1$ and $L2$ which are variables and not attributes If we treat actions as dummy nonterminals then such variables can be treated as the synthesized attributes of dummy nonterminals Since $L1$ and $L2$ do not depend on any other attributes they can be assigned to the rst action in the production The resulting SDT with embedded actions that implements this L attributed de nition is shown in Fig 5 28 □

S	**while**	$\{$ $L1$ *new* $L2$ *new* C *false* S *next* C *true* $L2$ $\}$
	C	$\{$ S_1 *next* $L1$ $\}$
	S_1	$\{$ S *code* **label** $\|\|$ $L1$ $\|\|$ C *code* $\|\|$ **label** $\|\|$ $L2$ $\|\|$ S_1 *code* $\}$

Figure 5 28 SDT for while statements

5 4 6 Exercises for Section 5 4

Exercise 5 4 1 We mentioned in Section 5 4 2 that it is possible to deduce from the LR state on the parsing stack what grammar symbol is represented by the state How would we discover this information

Exercise 5 4 2 Rewrite the following SDT

$$\begin{array}{ll} A & A\ \{a\}\ B \mid A\ B\ \{b\} \mid 0 \\ B & B\ \{c\}\ A \mid B\ A\ \{d\} \mid 1 \end{array}$$

so that the underlying grammar becomes non left recursive Here a b c and d are actions and 0 and 1 are terminals

Exercise 5 4 3 The following SDT computes the value of a string of 0 s and 1 s interpreted as a positive binary integer

$$
\begin{array}{lll}
B & & B_1\ 0\ \{B\ val \quad 2 \quad B_1\ val\} \\
 & | & B_1\ 1\ \{B\ val \quad 2 \quad B_1\ val \quad 1\} \\
 & | & 1\ \{B\ val \quad 1\}
\end{array}
$$

Rewrite this SDT so the underlying grammar is not left recursive and yet the same value of $B\ val$ is computed for the entire input string

Exercise 5 4 4 Write L attributed SDD s analogous to that of Example 5 19 for the following productions each of which represents a familiar ow of control construct as in the programming language C You may need to generate a three address statement to jump to a particular label L in which case you should generate **goto** L

a S **if** C S_1 **else** S_2

b S **do** S_1 **while** C

c S $'\{'\ L\ '\}'$ L $L\ S\ |$

Note that any statement in the list can have a jump from its middle to the next statement so it is not su cient simply to generate code for each statement in order

Exercise 5 4 5 Convert each of your SDD s from Exercise 5 4 4 to an SDT in the manner of Example 5 19

Exercise 5 4 6 Modify the SDD of Fig 5 25 to include a synthesized attribute $B\ le$ the length of a box The length of the concatenation of two boxes is the sum of the lengths of each Then add your new rules to the proper positions in the SDT of Fig 5 26

Exercise 5 4 7 Modify the SDD of Fig 5 25 to include superscripts denoted by operator **sup** between boxes If box B_2 is a superscript of box B_1 then position the baseline of B_2 0 6 times the point size of B_1 above the baseline of B_1 Add the new production and rules to the SDT of Fig 5 26

5 5 Implementing L Attributed SDD s

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