MODULE – 5

Syntax-Directed Translation

Structure:

1.1 Syntax Directed Translation
1.2 Intermediate code generation
1.3 Code generation

1.1 **Syntax-directed translation (SDT**) refers to a method of compiler implementation where the source language translation is completely driven by the parser, i.e., based on the syntax of the language The parsing process and parse trees are used to direct semantic analysis and the translation of the source program. Almost all modern compilers are syntax-directed.

SDT can be a separate phase of a compiler or we can augment our conventional grammar with information to control the semantic analysis and translation. Such grammars are called **attribute grammars**.

We augment a grammar by associating **attributes** with each grammar symbol that describes its properties. With each production in a grammar, we give **semantic rules/ actions**, which describe how to compute the attribute values associated with each grammar symbol in a production.

The general approach to Syntax-Directed Translation is to construct a parse tree or syntax tree and compute the values of attributes at the nodes of the tree by visiting them in some order.

A class of syntax-directed translations called "L- ttrib ted translations" (L for left-to-right) includes almost all translations that can e performed during parsing. Similarly, "S-attributed translations" (S for synthesized) c n be performed easily in connection with a bottom-up parse.

There are two ways to represent the emantic rules associated with grammar symbols.

- Syntax-Directed Definitions (SDD)
- Syntax-Directed Transl tion Schemes (SDT)

A syntax-directed definition (SDD) is a context-free grammar together with attributes and rules. Attributes are associated with grammar symbols and rules are associated with productions.

An attribute has a name and an associated value: a string, a number, a type, a memory location, an assigned register, strings. The strings may even be long sequences of code, say code in the intermediate language used by a compiler. If X is a symbol and a is one of its attributes, then we write X.a to denote the value of a at a particular parse-tree node labeled X. If we implement the nodes of the parse tree by records or objects, then the attributes of X can be implemented by data fields in the records that represent the nodes for X. The attributes are evaluated by the semantic rules attached to the productions.

SDDs are highly readable and give high-level specifications for translations. But they hide many implementation details. For example, they do not specify order of evaluation of semantic actions.

yntax-Directed Translation Schemes (SDT)

SDT embeds program fragments called semantic actions within production bodies. The position of semantic action in a production body determines the order in which the action is executed.

Example: In the rule E → E1 + T { print '+' }, the action is positioned after the body of the production.

SDTs are more efficient than SDDs as they indicate the order of evaluation of semantic actions associated with a production rule. This also gives some information about implementation details.

Inherited and Synthesized Attributes

Terminals can have synthesized attributes, which are given to it by the lexer (not the parser). There are no rules in an SDD giving values to attributes for terminals. Terminals do not have inherited attributes.

A nonterminal A can have both inherited and synthesized attributes. The difference is how they are computed by rules associated with a production at a node N of the parse tree.

- A **synthesized attribute** for a nonterminal A at a par e-tree node N is defined 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 defined only in terms of attribute values at the children of N and at N itself.

- An **inherited attribute** for nonterminal B at a parse-tree node N is defined by a semantic rule associated with the production at the parent of N. Note that the production must have B as a ymbol in its body.

  An inherited attribute at node N is defined only in terms of attribute values at N's parent, N itself, nd N's siblings.

An inherited attribute at node N cannot be defined in terms of attribute values at the children of node N. Ho ever, a synthesized attribute at node N can be defined in terms of inherited attribute values at node N itself.

Production	Semantic Rules
1) $L \rightarrow E$ **n**	$L.val = E.val$
2) $E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
3) $E \rightarrow T$	$E.val = T.val$
4) $T \rightarrow T_1 * F$	$T.val = T_1.val \times F.val$
5) $T \rightarrow F$	$T.val = F.val$
6) $F \rightarrow (E)$	$F.val = E.val$
7) $F \rightarrow$ **digit**	$F.val =$ **digit**$.lexval$

Fig 5.1: Syntax Directed Definition of simple desk calculator

.

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.

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.

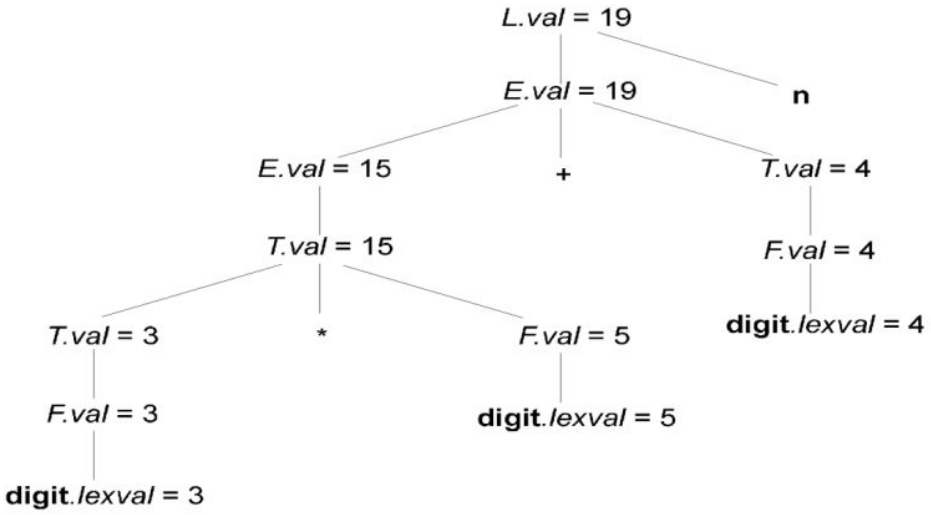
Attribute Grammar: An SDD without side effects is sometimes called an attribute grammar. The rules in an attribute grammar define the value of an attribute purely in terms of the values of other attributes and constants.

Evaluating an SDD at the Nodes of a Parse Tree

Parse tree helps us to visualize the translation specified by SDD. The rules of an SDD are applied by first 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.**

With synthesized attributes, we can evaluate attributes in any bottom-up order, such as that of a postorder traversal of the parse tree.

Example: Annotated Parse Tree for 3*5+4n



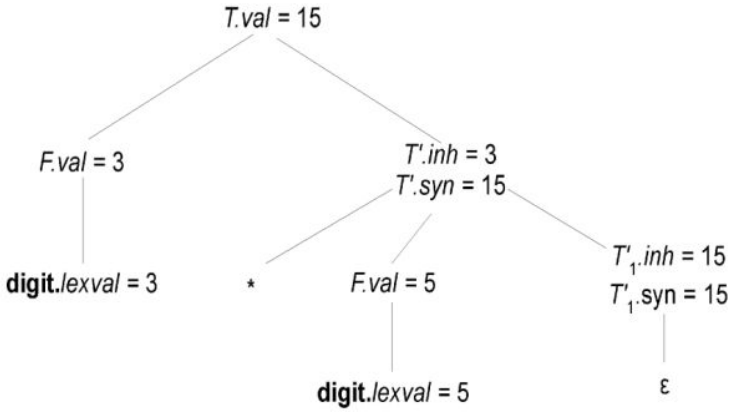
Inherited attributes are useful when the structure of a parse tree does not match the abstract syntax of the source code. They can be used to overcome the mismatch due to grammar designed for parsing rather than translation.

In the SDD below, the nonterminal T has an inherited attribute *inh* as well as a synthesized attribute val. T inherits F.val from its left sibling F in the production T → F T .

Production	Semantic Rules
$T \rightarrow F\ T'$	$T'.inh = F.val$ $T.val = T'.syn$
$T' \rightarrow *F\ T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
$T' \rightarrow \varepsilon$	$T'.syn = T'.inh$
$F \rightarrow$ **digit**	$F.val =$ **digit**$.lexval$

SDD for expression grammar with inherited attributes

Annotated Parse Tree for 3*5 using the above SDD is as below.



An SDD with both inherited and synthesized attributes does not ensure any guaranteed

order; even it may not have an order at . For ex mple, consider nonterminals A and B,

with synthesized and inherited attributes A. and B.i, respectively, along with the production and rules as in Fig.5.2. The e rules are circular; it is impossible to evaluate either A.s at a node N or B.i at the child of N without first 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.

e.g.

Production	Semantic Rules
$A \rightarrow B$	$A.s = B.i$ $B.i = A.s + 1$

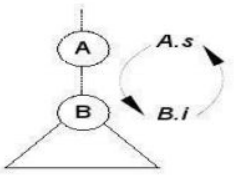


Fig 5.2: The circular dependency of *A.s* and *B.i* on one another

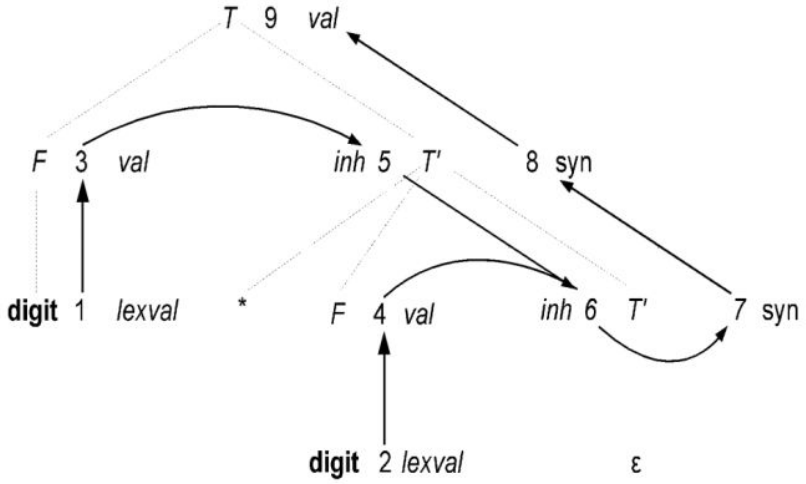
Evaluation Orders for SDD's

"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.

A dependency graph shows the flow of information among the attribute instances in a particular parse tree; an edge from one attribute instance to another means that the value of the first is needed to compute the second. Edges express constraints implied by the semantic rules.

- Each attribute is associated to a node
- If a semantic rule associated with a production *p defines the* value of synthesized attribute *A.b in terms of the value of X.c, then graph has an edge from X.c to A.b*
- If a semantic rule associated with a production *p defines the* value of inherited attribute *B.c in terms of value of X.a, then* graph has an edge from *X.a to B.c*

Example: Dependency graph for the annotated parse tree for 3*5

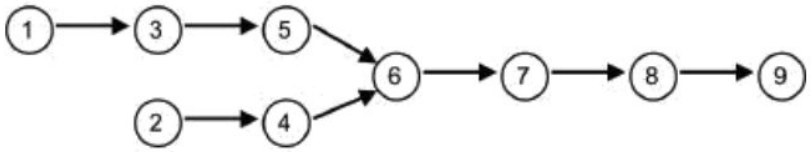


Topological Sort of the Dependency Graph

A dependency graph characterizes the possible order in which we can evaluate the attributes at various nodes of p rse tree. If there is an edge from node M to N, then attribute corresponding to M first be evaluated before evaluating N. Thus the only allowable orders of evaluation are N_1, N_2, ..., N_k such that if there is an edge from N_i to N_j then $i < j$. 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.

Example: Topological sorts for the above the dependency graph is shown in a simplified fashion with the nodes indicating the nodes of the parse tree and the edges indicating precedence of evaluation. It is apparent that the last section of the dependence graph denoted by the nodes { 6 7 8 9 } needs to be evaluated in sequence. The only flexibility of the order of evaluation is on the two sub-sequences { 1 3 5 } and { 2 4 } which can with any interleaving provided the relative order of their nodes is preserved are as below.



Base sequences are {1 3 5} and {2 4} with the suffix {6 7 8 9} being constant as the last nodes of the topological sorting need to remain fixed.
1 3 5 2 4 6 7 8 9, 1 3 2 5 4 6 7 8 9, 1 2 3 5 4 6 7 8 9, 1 3 2 4 5 6 7 8 9, 1 2 3 4 5 6 7 8 9,
1 2 4 3 5 6 7 8 9, 2 1 3 5 4 6 7 8 9, 2 1 3 4 5 6 7 8 9, 2 1 4 3 5 6 7 8 9, 2 4 1 3 5 6 7 8 9

S-Attributed Definitions

An SDD is *S-attributed* if every attribute is synthesized. Attributes of an S-attributed SDD can be evaluated in bottom-up order of the nodes of parse tree. Evaluation is simple using post-order traversal.

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

S-attributed definitions can be implemented during ottom-up parsing as

- bottom-up parse corresponds to postorder traversal
- postorder corresponds to the order in which an LR parser reduces a production body to its head

L-Attributed Definitions

The idea behind this class is th t, 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"). Each attribute must be either

1. Synthesized, or

2. Inherited, but ith the rules limited as follows. Suppose that there is a production $A \rightarrow X1\ X2 \bullet \bullet \bullet Xn$, and that there is an inherited attribute $Xi.a$ 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 occurrences of symbols *X1 X2 • • • Xi-1* located to the left of *Xi*

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

Example 1: The following definition is L-attributed. Here the inherited attribute of T gets its values from its left sibling F. Similarly, T1 gets its value from its parent T and left sibling F.

Production	Semantic Rules
$T \rightarrow F\ T'$	$T'.inh = F.val$
$T' \rightarrow *FT_1'$	$T_1'.inh = T'.inh \times F.val$

Example 2: the definitions below are not L-attributed as B.i depends on its right sibling C's attribute.

Production	Semantic Rules
$A \rightarrow B\ C$	$A.s = B.b$
	$B.i = f(C.c, A.s)$

Side Effects: Evaluation of semantic rules may generate intermediate codes, may put information into the symbol table, may perform type checking and may issue error messages. These are known as side effects.

Semantic Rules with Controlled Side Effects:

In practice translation involves side effects. Attribute grammars have no side effects and allow any evaluation order consistent with dependency graph whereas translation schemes impose left-to-right evaluation and allow schematic actions to contain any program fragment.

Ways to Control Side Effects

1. Permit incidental side effects that do not disturb attribute evaluation.
2. Impose restrictions on allowable evaluation order , o that the same translation is produced for any allowable order.

SDD For Simple Type Declarations

Production	Semantic Rules
1) $D \rightarrow T\ L$	$L.inh = T.type$
2) $T \rightarrow$ **int**	$T.type$ = integer
3) $T \rightarrow$ **float**	$T.type$ = float
4) $L \rightarrow L_1$, **id**	$L_1.inh = L.inh$ *addType* (**id**.*entry, L.inh*)
5) $L \rightarrow$ **id**	*addType* (**id**.*entry, L.inh*)

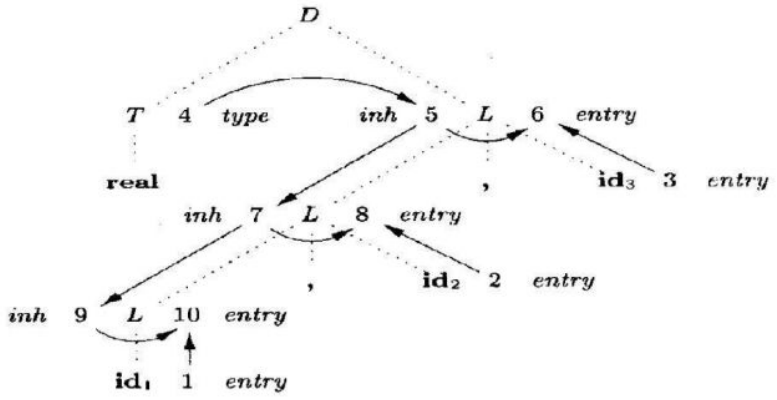
Nonterminal *D* represents a declaration, which, from production 1, consists of a type *T* followed by a list *L* of identifiers. *T* has one attribute, *T.type,* which is the type in the declaration *D*. Nonterminal *L* also has one attribute, which we call *ink* to emphasize that it is an inherited attribute. The purpose of *L.inh* is to pass the declared type down the list of identifiers, so that it can be the appropriate symbol-table entries. Productions 2 and 3 each evaluate the synthesized attribute *T.type,* giving it the appropriate value, integer or float. 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 L1.*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 identifier on the list.

The function *addType* properly installs the type *L.inh* as the type of the represented identifier. Note that the side effect, adding the type info to the table, does not affect the evaluation order.

A dependency graph for the input string **float id1 , id 2, id3** is shown below.



Applications of Syntax-Directed Translations

1: Construction of Syntax Trees

SDDs are useful for is construction of synt x trees. A syntax tree is a condensed form of parse tree.

- Syntax trees are useful for representing programming language constructs like expressions and statements.
- They help compiler design by decoupling parsing from translation.

- Each node of a syntax tree represents a construct; the children of the node represent the meaningful components of the construct.
  - e.g. a syntax-tree node representing an expression *E1 + E2 has* label + and two children representing the sub expressions *E1 and E2*
- Each node is implemented by objects with suitable number of fields; each object will have an *op field that is the label of the node with* additional fields as follows:

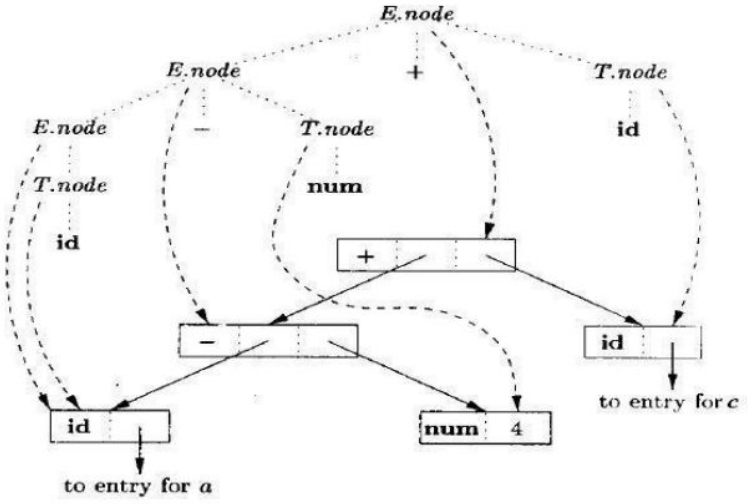
If the node is a leaf, an additional field holds the lexical value for the leaf . This is created by function **Leaf(op, val)**

If the node is an interior node, there are as many fields as the node has children in the syntax tree. This is created by function **Node(op, c1, c2,...,ck)** .

Example: The S-attributed definition in figure below 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.

Production	Semantic Rules
1) $E \rightarrow E_1 + T$	*E.node* = **new** *Node* ('+', E_1.*node*, *T.node*)
2) $E \rightarrow E_1 - T$	*E.node* = **new** *Node* ('−', E_1.*node*, *T.node*)
3) $E \rightarrow T$	*E.node* = *T.node*
4) $T \rightarrow (E)$	*E.node* = *T.node*
5) $T \rightarrow$ **id**	*T.node* = **new** *Leaf* (**id**, **id**.*entry*)
6) $T \rightarrow$ **num**	*T.node* = **new** *Leaf* (**num**, **num**.*val*)

Syntax tree for a-4+c using the above SDD is shown below.



Steps in the construction of the syntax tree for a-4+c

If the rules are evaluated during a post order traversal of the parse tree, or with reductions during a bottom-up parse, then the sequence of steps shown below ends with p5 pointing to the root of the constructed syntax tree.

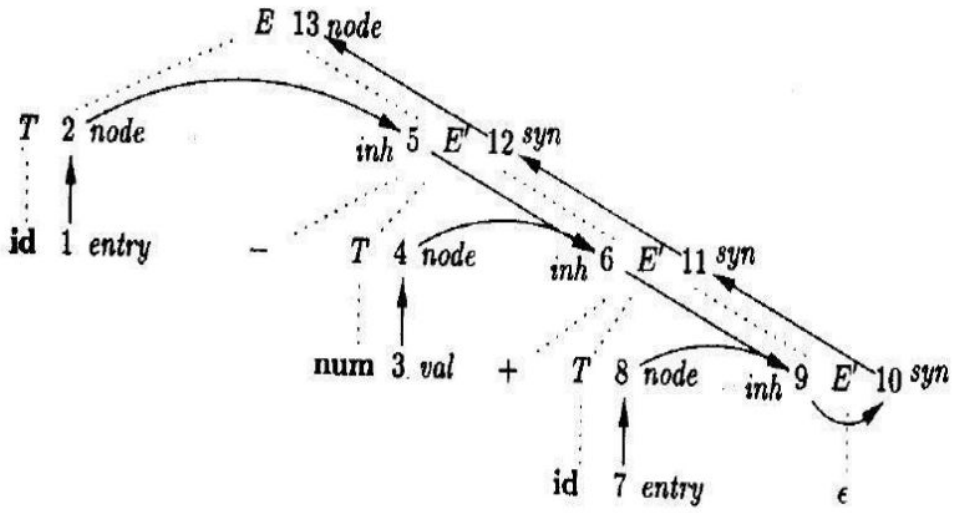
1) p_1 = **new** *Leaf* (**id**, *entry-a*);
2) p_2 = **new** *Leaf* (**num**, 4);
3) p_3 = **new** *Node* ('−', p_1, p_2);
4) p_4 = **new** *Leaf* (**id**, *entry-c*);
5) p_5 = **new** *Node* ('+', p_3, p_4);

Constructing Syntax Trees during Top-Down Parsing

With a grammar designed for top-down parsing, the same syntax trees areconstructed, using the same sequence of steps, even though the structure of the parse trees differs significantly from that of syntax trees. The L-attributed definition below performs the same translation as the S-attributed definition shown before.

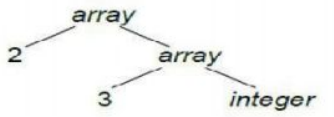
Production	Semantic Rules
1) $E \rightarrow T\ E'$	*E.node* = *E'.syn* *E'.inh* = *T.node*
2) $E' \rightarrow +T\ E_1'$	E_1'*.inh* = **new** *Node* ('+', *E'.inh*, *T.node*) *E'.syn* = E_1'*.syn*
3) $E' \rightarrow -T\ E_1'$	E_1'*.inh* = **new** *Node* ('−', *E'.inh*, *T.node*) *E'.syn* = E_1'*.syn*
4) $E' \rightarrow \varepsilon$	*E'.syn* = *E'.inh*
5) $T \rightarrow (\ E\ )$	*T.node* = *E.node*
6) $T \rightarrow$ **id**	*T.node* = **new** *Leaf* (**id**, **id**.*entry*)
7) $T \rightarrow$ **num**	*T.node* = **new** *Leaf* (**num**, **num**.*val*)

Dependency Graph for a-4+c with L-Attributed SDD



Structure of a Type

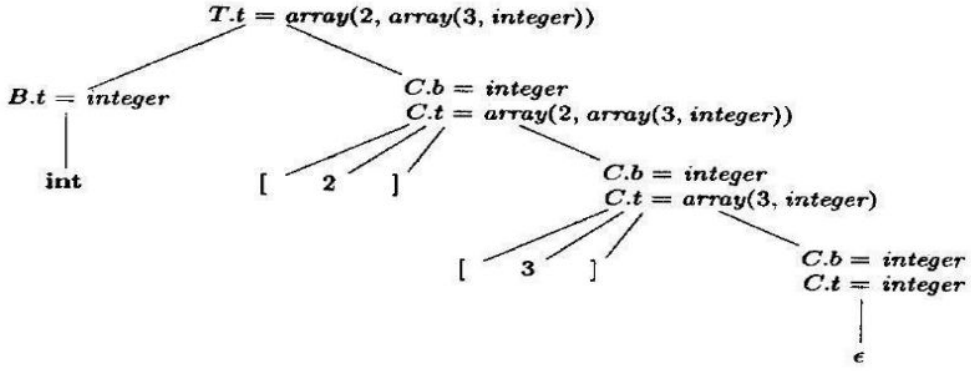
This is an example of how inherited attributes can be used to carry information one part of the parse tree to another. In C, the type int [2][3] can be read as, "array of 2 arrays of 3 integers." The corresponding type expression array(2, array(3, integer)) is represented by the tree as shown below.



Production	Semantic Rules
1) $T \rightarrow B\ C$	$T.t = C.t$ $C.b = B.t$
2) $B \rightarrow$ **int**	$B.t =$ *integer*
3) $B \rightarrow$ **float**	$B.t =$ *float*
4) $C \rightarrow$ **[num]** C_1	$C.t =$ *array* (**num**.*val*, $C_1.t$) $C_1.b = C.b$
5) $C \rightarrow \varepsilon$	$C.t = C.b$

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 below. The corresponding type expression 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.



1.2. Intermediate Code Generation

Introduction

In the first pass of the compiler, source program is converted into intermediate code. The second pass converts the intermediate code to target code. The intermediate code generation is done by intermediate code generation phase. It takes input from front end which consists of lexical analysis, syntax analysis and semantic analysis and generates intermediate code and gives it to code generator. Fig 6.1 shows the position of intermediate code generator in compiler. Although some source code can be directly converted to target code, there are some advantages of intermediate code. Some of these advantages are:

Target code can be generated to any machine just by attaching new machine as the back end. This is called retargeting.

It is possible to apply machine independent code optimization. This helps in faster generation of code.

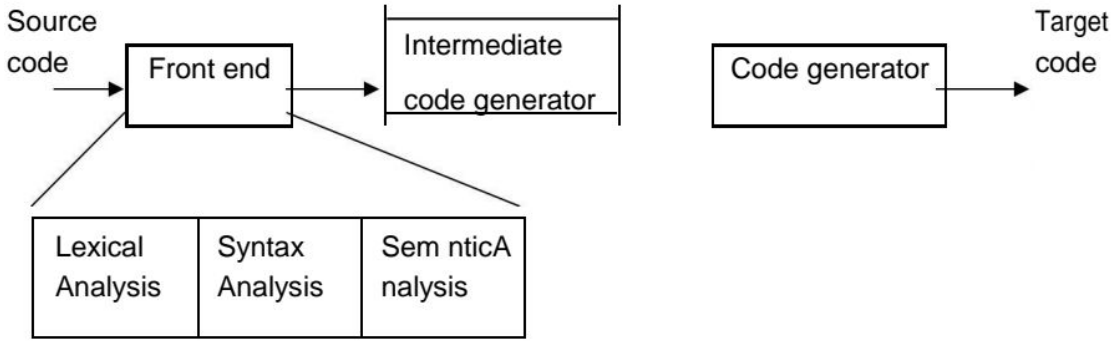


Fig 6.1 position of intermediate code generator in compiler

This chapter deals with the intermediate representation in the form of three address code. Other forms of intermediate representations are syntax tree, postfix notation or Directed Acyclic Graph (DAG). The semantic rule for syntax tree and three address code are almost similar.

Graphical and Linear representation

Intermediate representation can be either in linear or graphical form. Graphical form includes syntax tree and DAG whereas linear representation may be postfix notation or three address code. Fig 6.2 shows the syntax tree for the expression a = b * c.

Syntax tree

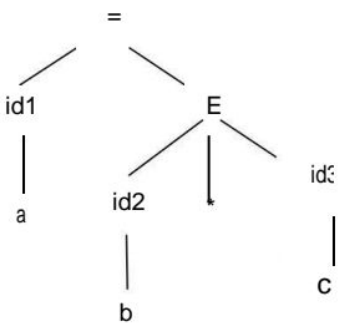


Fig 6.2 syntax tree for the expression a = -b * c

Directed Acyclic Graph (DAG)

For Example a = -b * C

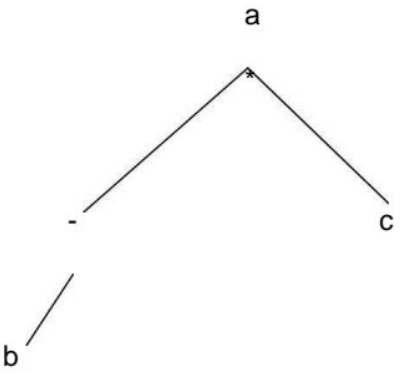


Fig: DAG for the expression a = -b * c

Linear representation: Postfix and three address code are the two forms of linear representation of any expression.

Postfix notation for the expression a = -b * c

is b-c*a= Three address code for the

expression a = - b * c is

t1= -b
t2=t*c
a=b

Intermediate code can be of many forms. They can be either

Language specific like P-code for Pascal, byte code for Java etc or

Specific to machine on which implementation is performed or

Independent of language being implemented and target machine

The three address code that we are considering here for intermediate representation is Independent of language being implemented and target machine.

Three address code

Most instruction of three address code is of the form

a = b op c

where, b and c are operands and op is an operator. The result after applying operator op on b and c is stored in a. Operator op can be like +, -, * or |. Here operator op is assumed as binary operator. The operands b and c represents the address in memory or can be con tants or literal value with no runtime address. Result a can be address in memory or temporary variable.

Example: a = b * c + 10

The three address code will be

t1 = b * C

t2 = t1 + 10

a = t2

Here t1 and t2 are temporary variables used to store the intermediate result.

Types of three address code

There are different types of statements in source program to which three address code has to be generated. Along with operands and operators, three address code also use labels to provide flow of control for statements like if-then-else, for and while. The different types of three address code statements are:

i. Assignment statement a = b op c

In the above case b and c are operands, while op is binary or logical operator. The result of applying op on b and c is stored in a.

ii. Unary operation a = op b

This is used for unary minus or logical negation.

Example: a = b * (- c) + d

Three address code for the above example will be

t1 = -c

t2 = t1 * b

t3 = t2 + d

a = t3

iii. Copy Statement a = b

The value of b is stored in variable .

iv. Unconditional jump goto L

Creates label L and generates three-address code 'goto L'

v. Conditional jump if exp go to L

Creates label L, generate code for expre sion exp, If the exp returns value true then go to the statement labelled L. exp returns value false go to the statement immediately following the if statement.

Function call For a function fun with n arguments a1,a2,a3….an

ie., fun(a1, a2, a3,…an),

the three address code will

be Param a1

Param a2

…

Param an

Call fun, n

Where param defines the arguments to function.

- Array indexing- In order to access the elements of array either single dimension or multidimension, three address code requires base address and offset value. Base address consists of the address of first element in an array. Other elements of the array can be accessed using the base address and offset value.

Example: x = y[i]

Memory location m = Base address of y + Displacement i

x = contents of memory location m

similarly x[i] = y

Memory location m = Base address of x + Displacement i

The value of y is stored in memory location m

viii. Pointer assignment x = &yxstorestheaddre of memory location y

x = *y y is a pointer whose r-value is location

*x = y sets r-value of the object pointed by x to the r-value of y

Intermediate representation should have an operator set which is rich to implement most of the operations of source language. It shou d so help in mapping to restricted instruction set of target machine.

Data Structure

Three address code is represented as record structure with fields for operator and operands. These records can be stored as array or linked list. Most common implementations of three address code are-Quadruples, Triples and Indirect triples.

Quadruples- Quadruples consists of four fields in the record structure. One field to store operator op, two fields to store operands or arguments arg1and arg2 and one field to store result res. res = arg1 op arg2

Example: a = b + c

b is represented as arg1, c is represented as arg2, + as op and a as res.

Unary operators like '-'do not use agr2. Operators like param do not use agr2 nor result. For conditional and unconditional statements res is label. Arg1, arg2 and res are pointers to symbol table or literal table for the names.

Example: a = -b * d + c + (-b) * d

Three address code for the above statement is as follows

t1 = - b

t2 = t1 * d

t3 = t2 + c

t4 = - b

t5 = t4 * d

t6 = t3 + t5

a = t6

Quadruples for the above example is as follows

Op	Arg1	Arg2	Res
-	B		t1
*	t1	d	t2
+	t2	c	t3
-	B		t4
*	t4	d	t5
+	t3	t5	t6
=	t6		a

Triples – Triples uses only three fields in the record structure. One field for operator, two fields for operands named as arg1 and arg2. Value of temporary variable can be accessed by the position of the statement the computes it and not by location as in quadruples.

Example: a = -b * d + c + (-b) * d

Triples for the above example is as follows

Stmt no	Op	Arg1	Arg2
(0)	-	b	
(1)	*	d	(0)
(2)	+	c	(1)
(3)	-	b	
(4)	*	d	(3)
(5)	+	(2)	(4)
(6)	=	a	(5)

Arg1 and arg2 may be pointers to symbol table for program variables or literal table for constant or pointers into triple structure for intermediate results.

Triples are alternative ways for representing syntax tree or Directed acyclic graph for program defined names.

Indirect Triples – Indirect triples are used to achieve indirection in listing of pointers. That is, it uses pointers to triples than listing of triples themselves.

Basic Intermediate Code Generation Technique

Program consists of assignment statements like a=b op c or control statements like if-then-else, while loop or for statements. This section deals with generation of three address code for assignment statement and control statements.

Assignment

This section deals with the generation of intermediate code for assignment statement. It describes the way in which symbol table can be searched for an identifier. Identifiers can be simple variable or single or multidimensional array or a constant value (stored in literal table). Next step is generation of three address code for the program statement.

- Searching in symbol table

During the process of generation of intermediate code, symbol table has to be searched for identifier. The lexeme for identifier is stored in variable id.name. Searching for identifier in symbol table is achieved through the function search(). Search() returns the pointer of identifier in symbol table, if id.name has an entry in symbol table. If search fails it returns null indicating id.name not found.

- Generate code

Intermediate code generator uses function called produce() to generate three address code and store it in output file. It also uses a variable E.value to store the name of E that holds the vale of E. All the intermediate results are to be stored intemporaryvariables.Hence the function newtemp() is used. It generates new temporary variables like t1,t2,… ever time newtemp() is called.

Example: Consider the following grammar for assignment statement

S → id=E

E → E1 + E2

E → E1 * E2

E → -E1

E → (E1)

E → id

Translation scheme to produce three address code is as follows

Production	Translation rules
S → id=E	x = search(id.name); if x ≠ null then produce(x '=' E.value) else error

E → E1 + E2	E.value = newtemp() Produce(E.value '=' E1.value '+' E2.value)
E → E1 * E2	E.value = newtemp() Produce(E.value '=' E1.value '*' E2.value)
E → -E1	E.value = newtemp() Produce(E.value '=' '-' E1.value)
E → (E1)	E.value = E1.value
E → id	x = search(id.name) if x ≠ null then E.value = x else error

Example: Generate three address code or the following rithmetic expression

a = -b * c

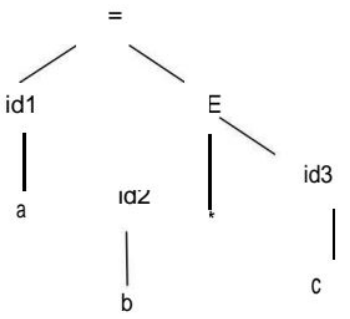


Fig 6.4 Syntax tree for the expression a = -b * c

Consider the Fig 6.4 which shows the syntax tree for the expression a = -b * c

- Using the production E → id i.e.. E → b Check for entry b in the symbol table if not present display error msg., if present made E.value = x i.e., E.value = b (pointer to symbol table for the entry b)
- Create a temporary variable t_1 with production 4, this produces an intermediate code

$t1 = -b$

7. Same as step 1, searches for entry c in symbol table & assigns E.value = c (pointer to symbol table for the entry c)

8. Using the production $E \rightarrow E_1 * E_2$ it creates temporary variable t2 using newtemp(). Three address code $E.value = E_1.value + E_2.value$ generates

$t_2 = t_1 + c$

- Using the production $S \rightarrow id = E$ searches for a in symbol table, assuming it is stored produces code x = E.value

$a = t_2$

Reusing temporary variables

The newtemp() function is used to createnewtemporaryvariableto store intermediate results. If the number of temporary variables increases to a large val e, it becomes difficult to maintain these

variables. In some cases the temporary variables values may not be required until the end. So, the memory reserved for theses variables goes unused, in order to use memory efficiently and reduce the number of temporary variables, reusing of temporary variables is done. The newtemp() function has to be modified to achieve this. Instead of alwa s generating new temporary variables every time a newtemp() is called, it should find those temporary variable whose use is completed in the subsequent code, then the intermediate results can be stored i these temporary variables ie they are reused.

The other significance of reusing temporary is that, there can be some sub expressions being repeated in the whole expression. Instead of creating new temporary variable for all sub expressions, it can be reused.

Example: a = -b * c + (-b) * C

The three address code will be as follows:

7. $t_1 = -b$
8. $t_2 = t_1 + c$
9. $t_3 = t_2 + d$
10. $a = t_3$

after the statement (2) t_1 is not used hence instead of creating new. Temporary t_3, t_1 can be reused. Hence generates following code

⇓

- $t_1 = - b$
- $t_2 = t_1 + c$
- $t_1 = t_2 + d$
- $a = t_1$

number of temporary variables used is 2 instead of 3

Example:

(1)	$t_1 = - b$		(1)	$t_1 = - b$
(4)	$t_4 = t_3 * c$		(4)	$= t_5$ com
(2)	$t_2 = t_1 * c$		(2)	$t_2 = t_1 *$
(3)	$t_3 = - b$	can be changed to	(3)	$t_3 = t_2 + t_2$

8. $t_5 = t_2 + t_4$

9. $a = t_5$

Reducing number of temporary variablesfrom5to3andreusingvalue of t_2

1.3 Code Generation

Code generator phase generates the target code taking input as intermediate code. The output of intermediate code generator may be given directly to code generation or may pass through code optimization before generating code.

Issues in Design of Code generation:

Target code mainly depends on available instruction set and efficient usage of registers. The main issues in design of code generation are

Intermediate representation: Linear representation like postfix and three address code or quadruples and graphical representation like Syntax tree or DAG. Assume type checking is done and input in free of errors. This chapter deals only with intermediate representation as three address code.

- **Target Code:** The target code may be absolute code, re-locatable achine code or assembly language code. Absolute code can be executed immediately as the addresses are fixed. But in case of re-locatable it requires linker and loader to place the code in appropriate location and map (link) the required library functions. If it generates assembly level codethenassemblersareneeded to convert it into machine level code before execution. Re-locatable code provides great deal of flexibilities as the functions can be compiled separately before generation of object code.

- **Address mapping:** Address mapping defines the mapping between intermediate representations to address in the target code. These addresses are based on the runtime environment used like static, stack or heap. The identifiers are stored in symbol table during declaration of variables or functions, along with type. Each identifier can be accessed in symbol table based on width of each identifier and offset. The address of the specific instruction (in three ddress code) can be generated using back patching

- **Instruction Set**: The instruction set should be complete in the sense that all operations can be implemented. Some times a single operation may be implemented using many instruction (many set of instructions). The code generator should choose the most appropriate instruction. The instruction should be chosen in such a way that speed is of execution is minimum or other machine related resource utilization should be minimum.

Example: Consider the set of statements

a = b * c

d = a * e

Three address code will be as followssholu

$$t_1 = b * c$$

$$t_2 = t_1 + 10$$

$$t_3 = t_1 + t_2$$

Final code generated will be as follows

MOV b, R0 / load b to register Ro,

MUL C, R0

MOV.R0, a } Mov a to Ro and moving Ro to a can be eliminated

MOV a, R0 }

MUL e, R_0

MOV R_0, d

Redundant instruction should be eliminated.

Replace n instruction by single instruction

x = x + 1

MOV x, R_0 }

ADD 1, R_0 } ⇒INC x

MOV R_0. x }

Register allocation: If the operands are in register the executi n is faster hence the set of variables whose values are required at a point in the program are to be retained in the registers.

Familiarities with the target machineanditsinstructionsetare a pre-requisite for designing a good code generator.

Target Machine: Consider a hypothetical byte ddressable machine as target machine. It has n general purpose register R_1, R_2 ---------- R_n. The machine instructions are two address instructions of the form

op-code source address destination

addre Example:

MOV R_0, R_1

ADD R_1, R_2

Target Machine supports for the following addressing modes

Absolute addressing mode

Example: MOV R_0, M where M is the address of memory location of one of the operands. MOV R_0, M moves the contents of register R_0 to memory location M.

Register addressing mode where both the operands are in register. Example: ADD R_0, R_1

Immediate addressing mode – The operand value appears in the instruction. Example: ADD # 1, R_0

Index addressing mode- this is of the form C(R) where the address of operand is at the location C +Contents(R)

Example: MOV 4(R_0), M the operand is located at address = contents (4+contents(R_0))

Cost of instruction is defined as cost of execution plus the number of memory access.

Example:

MOV R_0, R_1, the cost = 1 as there are no memory

access. Where as MOV R_0, M cost = 2.

Register and address descriptor

Register descriptor gives the details of which values are stored in which registers and the list of registers which are free.

Address descriptor gives the location of the current value can be in register, memory location or stack based on runtime environment.

Code generation algorithm

Consider the simple three address code for which the target code to be generated.

Example: a = b op c

i. Consult the address descriptor for 'b' to find out whether b is in register or memory

location. If b is in memory ocation, generate code.

a. MOV b, R_i where R_i is one of the free register as per register descriptors.

Update address descriptor of and register descriptor for free registers.

ii. Generate code for OP C, here C can be in memory location or in register.

iii. Store result 'a' in location L. L can be memory location M or register R, based on availability of free register and further usage of 'a'. Update register descriptor and address descriptor for 'a' accordingly.

Example: x = y + z

Check for location of y,

Case 1: If y is in register R_0 and z may be in register or memory. The instructions will be

ADD z, R_0

MOV R_0, x In this case the result x has to be stored in memory location x.

MOV y, R_0

Case 2:

ADD z, R_0

MOV R_0, x

Example:

P = (x – y) + (x – z) + (x –

z) t_1 = x – y

t_2 = x – z

t_3 = t_1 +

t_2 t_4 = t_3

+ t_2

Three address code

3 addr	M/c Code	Cost	Reg desc	Addr desc
t_1 = x – y	MOV x , R_0 SUB y , R_0 =	2 2	R_0 has t_1	t_1 in R_0
t_2 = x – z	MOV x , R_1 SUB z, 12	2 2	R_0 has t_1 R_1 has t_2	T in R_0 U in R_1
t_3 = t_1 + t_2	ADD R_1 , R_0	1	R_0 has t_3 R_1 has t_2	t_2 in R_1 t_3 in R_0
t_4 = t_3 + t_2	ADD R_1, R_0 MOV R_0, t_4	1 2	R_0 has t_4	t_4 in R_0 and memory

Example: Generate code for instruction x = y[i] and x [i]=

Stmt	i in reg R_i		i in Memory		i in Stack	
	Code	Cost	Code	Cost	Code	Cost
x = y [i]	MOV y (R_i), R	2	MOV M, R MOV b (R1, R2)	4	MOV S_i (x), R MOV y (R), R	4
x [i] = y	MOV y, x (R_i)	3	MOV M, R	5	MOV S_i(x),x MOV y, x (R)	5

Code generation for function call

Code generation for function code is base on the runtime storage. The runtime storage can by static allocation or stack allocation. In case of static allocation the position of activation record in memory is fixed at the compile time. To recollect about activation record, whenever a function is called, activation records are generated, these records store the parameters to be passed to functions, local data, temporaries, results & some machine status information along with the return address. In case of stack allocation, every time a function is called, the new activation record in generated & is pushed onto stack, once the function completes, the activation record is popped from stack. The three address code for function call consists of following statements

- Call.
- Return
- end
- action

Call statement is used for function Call, it has to mail the control to the function along with saving the status of current function.

Return statement is used to give the control back to called function. Action defines other operations or instructions for assignment or flow control statements. End indicates the completion of operations of called function.

Static allocation: This section describes the final code generation for function calls, where static allocation is used as runtime environment.

- **Call statement :** The code generated for call stmt is as foll ws.

MOV # current + 20, function.static_area

GOTO function.code_area

current + 20 indicates the address of next instruction to which the return of function, i.e, the instruction of called function which has to be executed after the called function completes execution. 20 defines the size of goto statement fo lowing call stmt.

Function.static_area defines the address of activation record of function. Function.code_area defines the address of 1st instruction of called function.

- **Return Statement**: Code generated for return stmt is

goto * function.static area.

This allows the control back to the called function.

Example:

/* code for main */

action 1

call fun

action 2

end

/* code for fun */

action 3
return

Three address code that will be generated for the above set of statements is as follows.

10: action 1

20: MOV # 40, 200 /* Save return address 40 at location 200 */

30: GOTO 100

40: Action 2

50: end

/* code for function */

- action 3

100: GOTO * 200 200:

40(return address)

Stack allocation: Whenever the function is called the activation record of called fun c is stored on Stack, once the function returns, it is removed from Stack. Final code that will be generated for stack area for initialize the Stack i

```
MOV        # Stack.begin, SP      /* initialize the Stack Pointer */
SP denotes Stack Pointer.
```

Code for Call statement is as follows

```
Add # main.record size, SP       /*main.recordsize referes to
                                  record size of caller function*/

MOV # current +16, *SP           /*Save return address*/
GOTO function.code_area
```

Return statement has the following target code.

GOTO *0(SP)

SUB # main.recordsize, SP

Example: For the below three address code

/* code for a */

action1

call c

action 2

end