UNIT-3

SYNTAX DIRECTED TRANSLATION

In syntax directed translation, along with the grammar we associate some informal notations and these notations are called as semantic rules.

So, we can say that

1. Grammar + semantic rule = SDT (syntax directed translation)

- In syntax directed translation, every non-terminal can get one or more than one attribute or sometimes 0 attribute depending on the type of the attribute. The value of these attributes is evaluated by the semantic rules associated with the production rule.
- In the semantic rule, attribute is VAL and an attribute may hold anything like a string, a number, a memory location and a complex record
- In Syntax directed translation, whenever a construct encounters in the programming language then it is translated according to the semantic rules define in that particular programming language.

Example

Production	Semantic Rules
E → E + T	E.val := E.val + T.val
E → T	E.val := T.val
T → T * F	T.val := T.val + F.val
T → F	T.val := F.val
F → (F)	F.val := F.val
F → num	F.val := num.lexval

E.val is one of the attributes of E.

Syntax directed translation scheme

- The Syntax directed translation scheme is a context -free grammar.
- The syntax directed translation scheme is used to evaluate the order of semantic rules.
- In translation scheme, the semantic rules are embedded within the right side of the productions.
- The position at which an action is to be executed is shown by enclosed between braces. It is written within the right side of the production.

Example

Production	Semantic Rules
S → E $	{ printE.VAL }
E → E + E	{E.VAL := E.VAL + E.VAL }
E → E * E	{E.VAL := E.VAL * E.VAL }
E → (E)	{E.VAL := E.VAL }
E → I	{E.VAL := I.VAL }
I → I digit	{I.VAL := 10 * I.VAL + LEXVAL }
I → digit	{ I.VAL:= LEXVAL}

Implementation of Syntax directed translation

Syntax direct translation is implemented by constructing a parse tree and performing the actions in a left to right depth first order.

SDT is implementing by parse the input and produce a parse tree as a result.

Example

Production	**Semantic Rules**
S → E $	{ printE.VAL }
E → E + E	{E.VAL := E.VAL + E.VAL }
E → E * E	{E.VAL := E.VAL * E.VAL }
E → (E)	{E.VAL := E.VAL }
E → I	{E.VAL := I.VAL }
I → I digit	{I.VAL := 10 * I.VAL + LEXVAL }
I → digit	{ I.VAL:= LEXVAL}

Parse tree for SDT:

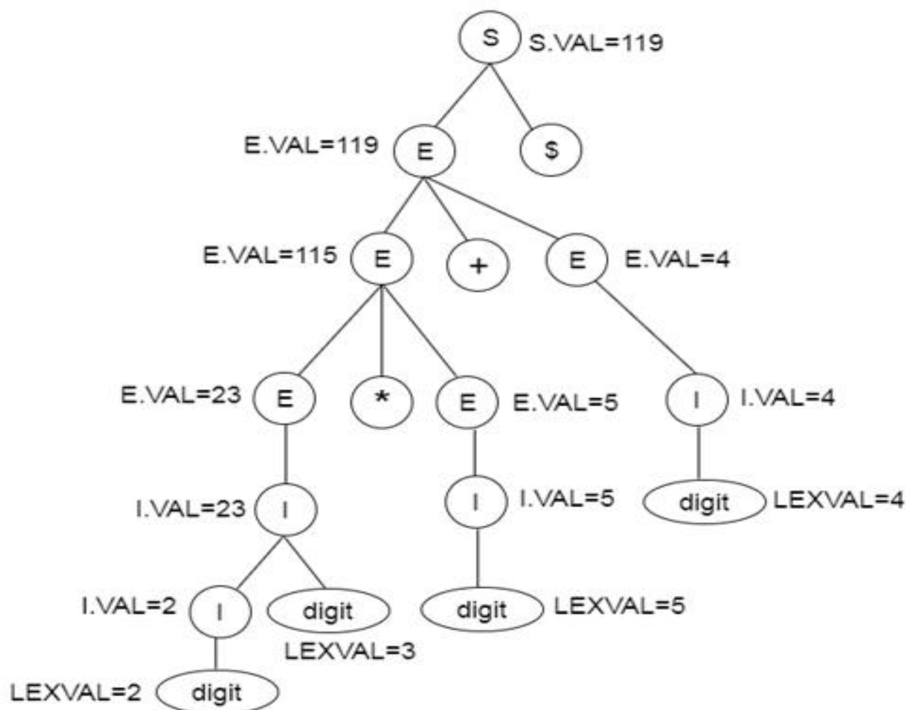


Fig: Parse tree

Intermediate code

Intermediate code is used to translate the source code into the machine code. Intermediate code lies between the high-level language and the machine language.

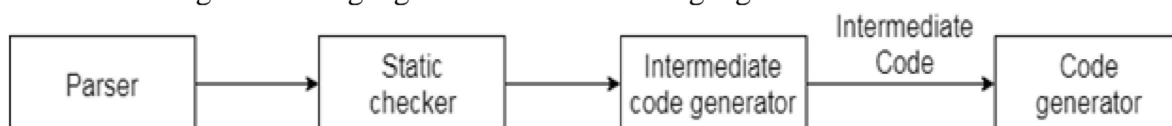


Fig: Position of intermediate code generator

- If the compiler directly translates source code into the machine code without generating intermediate code then a full native compiler is required for each new machine.
- The intermediate code keeps the analysis portion same for all the compilers that's why it doesn't need a full compiler for every unique machine.
- Intermediate code generator receives input from its predecessor phase and semantic analyzer phase. It takes input in the form of an annotated syntax tree.
- Using the intermediate code, the second phase of the compiler synthesis phase is changed according to the target machine.

Intermediate representation

Intermediate code can be represented in two ways:

1. High Level intermediate code:

High level intermediate code can be represented as source code. To enhance performance of source code, we can easily apply code modification. But to optimize the target machine, it is less preferred.

2. Low Level intermediate code

Low level intermediate code is close to the target machine, which makes it suitable for register and memory allocation etc. it is used for machine-dependent optimizations.

Postfix Notation

- Postfix notation is the useful form of intermediate code if the given language is expressions.
- Postfix notation is also called as 'suffix notation' and 'reverse polish'.
- Postfix notation is a linear representation of a syntax tree.
- In the postfix notation, any expression can be written unambiguously without parentheses.
- The ordinary (infix) way of writing the sum of x and y is with operator in the middle: x * y. But in the postfix notation, we place the operator at the right end as xy *.
- In postfix notation, the operator follows the operand.

Example

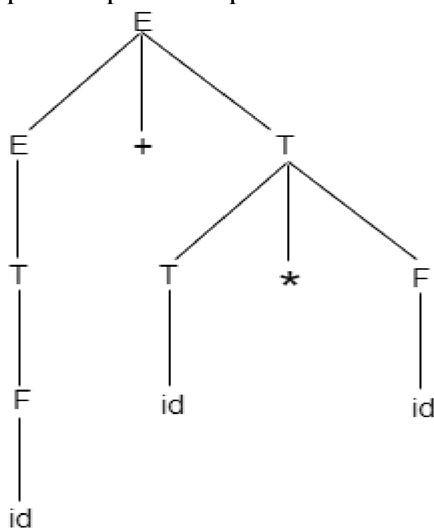
Production

1. E → E1 op E2
2. E → (E1)
3. E → id

Semantic Rule	**Program fragment**
E.code = E1.code || E2.code || op	print op
E.code = E1.code	
E.code = id	print id

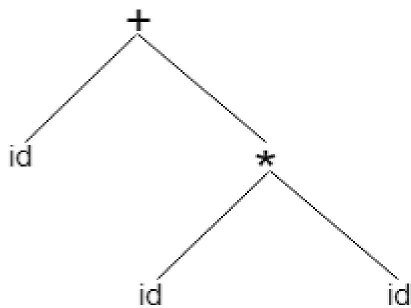
Parse tree and Syntax tree

When you create a parse tree then it contains more details than actually needed. So, it is very difficult to compiler to parse the parse tree. Take the following parse tree as an example:

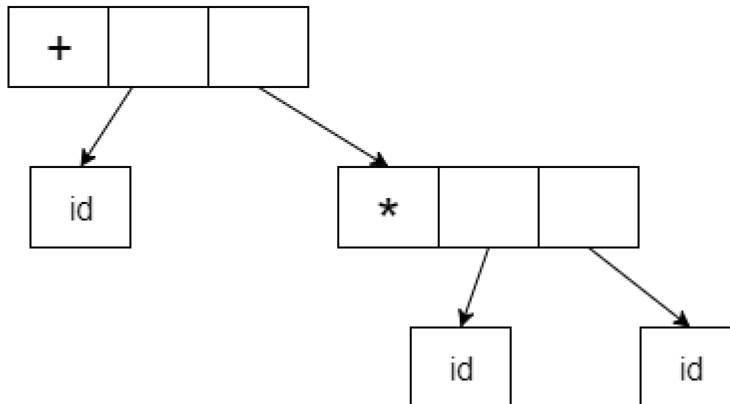


- In the parse tree, most of the leaf nodes are single child to their parent nodes.
- In the syntax tree, we can eliminate this extra information.
- Syntax tree is a variant of parse tree. In the syntax tree, interior nodes are operators and leaves are operands.
- Syntax tree is usually used when represent a program in a tree structure.

A sentence **id + id * id** would have the following syntax tree:



Abstract syntax tree can be represented as:



Abstract syntax trees are important data structures in a compiler. It contains the least unnecessary information.

Abstract syntax trees are more compact than a parse tree and can be easily used by a compiler.

Three address code

- Three-address code is an intermediate code. It is used by the optimizing compilers.
- In three-address code, the given expression is broken down into several separate instructions. These instructions can easily translate into assembly language.
- Each Three address code instruction has at most three operands. It is a combination of assignment and a binary operator.

Example
Given Expression:

1. a := (-c * b) + (-c * d)

Three-address code is as follows:

t_1 := -c t_2 := b*t_1 t_3 := -c t_4 := d * t_3 t_5 := t_2 + t_4 a := t_5

t is used as registers in the target program.

The three address code can be represented in two forms: **quadruples** and **triples**.

Quadruples
The quadruples have four fields to implement the three-address code. The field of quadruples contains the name of the operator, the first source operand, the second source operand and the result respectively.

Operator
Source 1
Source 2
Destination

Fig: Quadruples field

Example

1. a := -b * c + d

Three-address code is as follows:

t_1 := -b t_2 := c + d t_3 := t_1 * t_2 a := t_3

These statements are represented by quadruples as follows:

	Operator	**Source 1**	**Source 2**	**Destination**
(0)	uminus	b	-	t_1
(1)	+	c	d	t_2
(2)	*	t_1	t_2	t_3
(3)	:=	t_3	-	a

Triples

The triples have three fields to implement the three-address code. The field of triples contains the name of the operator, the first source operand and the second source operand.

In triples, the results of respective sub-expressions are denoted by the position of expression. Triple is equivalent to DAG while representing expressions.

Operator
Source 1
Source 2

Fig: Triples field

Example:

1. a := -b * c + d

Three address code is as follows:

t_1 := -b t_2 := c + dM t_3 := t_1 * t_2 a := t_3

These statements are represented by triples as follows:

	Operator	**Source 1**	**Source 2**
(0)	uminus	b	-
(1)	+	c	d
(2)	*	(0)	(1)
(3)	:=	(2)	-

Translation of Assignment Statements

In the syntax directed translation, assignment statement is mainly deals with expressions. The expression can be of type real, integer, array and records.

Consider the grammar

1. S → id := E
2. E → E1 + E2
3. E → E1 * E2
4. E → (E1)
5. E → id

The translation scheme of above grammar is given below:

Production rule	**Semantic actions**

S → id :=E	{p = look_up(id.name);
	If p ≠ nil then
	Emit (p = E.place)
	Else
	Error;
	}

E → E1 + E2	{E.place = newtemp(); Emit (E.place = E1.place '+' E2.place) }
E → E1 * E2	{E.place = newtemp(); Emit (E.place = E1.place '*' E2.place) }
E → (E1)	{E.place = E1.place}

E → id	{p = look_up(id.name);
	If p ≠ nil then
	Emit (p = E.place)
	Else
	Error;
	}

- The p returns the entry for id.name in the symbol table.
- The Emit function is used for appending the three-address code to the output file. Otherwise it will report an error.
- The newtemp() is a function used to generate new temporary variables.
- E.place holds the value of E.

Boolean expressions

Boolean expressions have two primary purposes. They are used for computing the logical values. They are also used as conditional expression using if-then-else or while-do.

Consider the grammar

1. E → E OR E
2. E → E AND E
3. E → NOT E
4. E → (E)
5. E → id relop id
6. E → TRUE
7. E → FALSE

The relop is denoted by <, >, <, >.

The AND and OR are left associated. NOT has the higher precedence then AND and lastly OR.

Production rule	**Semantic actions**
E → E1 OR E2	{E.place = newtemp(); Emit (E.place ':=' E1.place 'OR' E2.place) }
E → E1 + E2	{E.place = newtemp(); Emit (E.place ':=' E1.place 'AND' E2.place) }
E → NOT E1	{E.place = newtemp(); Emit (E.place ':=' 'NOT' E1.place) }
E → (E1)	{E.place = E1.place}
E → id relop id2	{E.place = newtemp(); Emit ('if' id1.place relop.op id2.place 'goto' nextstar + 3); EMIT (E.place ':=' '0') EMIT ('goto' nextstat + 2) EMIT (E.place ':=' '1') }
E → TRUE	{E.place := newtemp(); Emit (E.place ':=' '1') }
E → FALSE	{E.place := newtemp(); Emit (E.place ':=' '0') }

The EMIT function is used to generate the three address code and the newtemp( ) function is used to generate the temporary variables.

The E → id relop id2 contains the next_state and it gives the index of next three address statements in the output sequence.

Here is the example which generates the three-address code using the above translation scheme:

1. p>q AND r<s OR u>r
2. 100: **if** p>q **goto** 103
3. 101: t1:=0
4. 102: **goto** 104
5. 103: t1:=1
6. 104: **if** r>s **goto** 107
7. 105: t2:=0
8. 106: **goto** 108
9. 107: t2:=1
10. 108: **if** u>v **goto** 111
11. 109: t3:=0
12. 110: **goto** 112
13. 111: t3:= 1
14. 112: t4:= t1 AND t2
15. 113: t5:= t4 OR t3

Statements that alter the flow of control

The goto statement alters the flow of control. If we implement goto statements then we need to define a LABEL for a statement. A production can be added for this purpose:

1. S → LABEL : S
2. LABEL → id

In this production system, semantic action is attached to record the LABEL and its value in the symbol table.

Following grammar used to incorporate structure flow-of-control constructs:

1. S → **if** E then S
2. S → **if** E then S **else** S
3. S → **while** E **do** S
4. S → begin L end
5. S→ A
6. L→ L ; S
7. L → S

Here, S is a statement, L is a statement-list, A is an assignment statement and E is a Boolean-valued expression.

Translation scheme for statement that alters flow of control

- We introduce the marker non-terminal M as in case of grammar for Boolean expression.
- This M is put before statement in both if then else. In case of while-do, we need to put M before E as we need to come back to it after executing S.
- In case of if-then-else, if we evaluate E to be true, first S will be executed.
- After this we should ensure that instead of second S, the code after the if-then else will be executed. Then we place another non-terminal marker N after first S.

The grammar is as follows:

1. S → **if** E then M S
2. S → **if** E then M S **else** M S
3. S → **while** M E **do** M S
4. S → begin L end
5. S → A
6. L→ L ; M S
7. L → S
8. M → ∈
9. N → ∈

The translation scheme for this grammar is as follows:

Production rule	Semantic actions
S → if E then M S1	BACKPATCH (E.TRUE, M.QUAD) S.NEXT = MERGE (E.FALSE, S1.NEXT)
S → if E then M1 S1 else M2 S2	BACKPATCH (E.TRUE, M1.QUAD) BACKPATCH (E.FALSE, M2.QUAD) S.NEXT = MERGE (S1.NEXT, N.NEXT, S2.NEXT)
S → while M1 E do M2 S1	BACKPATCH (S1,NEXT, M1.QUAD) BACKPATCH (E.TRUE, M2.QUAD) S.NEXT = E.FALSE GEN (goto M1.QUAD)
S → begin L end	S.NEXT = L.NEXT
S → A	S.NEXT = MAKELIST ()
L → L ; M S	BACKPATHCH (L1.NEXT, M.QUAD) L.NEXT = S.NEXT
L → S	L.NEXT = S.NEXT
M → ∈	M.QUAD = NEXTQUAD
N→ ∈	N.NEXT = MAKELIST (NEXTQUAD) GEN (goto_)

Postfix Translation

In a production A → α, the translation rule of A.CODE consists of the concatenation of the CODE translations of the non-terminals in α in the same order as the non-terminals appear in α.

Production can be factored to achieve postfix form.

Postfix translation of while statement

The production

1. S → **while** M1 E **do** M2 S1

Can be factored as:

1. S → C S1
2. C → W E **do**
3. W → **while**

A suitable transition scheme would be

Production Rule	Semantic Action
W → while	W.QUAD = NEXTQUAD
C → W E do	C W E do

S→ C S1	BACKPATCH (S1.NEXT, C.QUAD) S.NEXT = C.FALSE GEN (goto C.QUAD)

Postfix translation of for statement

The production

1. S **for** L = E1 step E2 to E3 **do** S1

Can be factored as

1. F → **for** L
2. T → F = E1 by E2 to E3 **do**
3. S → T S1

Array references in arithmetic expressions

Elements of arrays can be accessed quickly if the elements are stored in a block of consecutive location. Array can be one dimensional or two dimensional.

For one dimensional array:

1. A: array [low..high] of the ith elements is at:
2. base + (i-low)*width → i*width + (base - low*width)

Multi-dimensional arrays:

Row major or column major forms

- Row major: a[1,1], a[1,2], a[1,3], a[2,1], a[2,2], a[2,3]
- Column major: a[1,1], a[2,1], a[1, 2], a[2, 2],a[1, 3],a[2,3]
- In raw major form, the address of a[i1, i2] is
- Base+((i1-low1)*(high2-low2+1)+i2-low2)*width

Translation scheme for array elements

Limit(array, j) returns nj=highj-lowj+1

place: the temporaryor variables

offset: offset from the base, null if not an array reference

The production:

1. S → L := E
2. E → E+E
3. E → (E)
4. E → L
5. L → Elist]
6. L → id
7. Elist → Elist, E
8. Elist → id[E

A suitable transition scheme for array elements would be:

Production Rule	Semantic Action

S → L := E	{if L.offset = null then emit(L.place ':=' E.place)
	else EMIT (L.place'['L.offset ']' ':=' E.place);
	}

E → E+E	{E.place := newtemp; EMIT (E.place ':=' E1.place '+' E2.place); }
E → (E)	{E.place := E1.place;}

E → L	{if L.offset = null then E.place = L.place
	else {E.place = newtemp;
	EMIT (E.place ':=' L.place '[' L.offset ']');
	}
	}

L → Elist]	{L.place = newtemp; L.offset = newtemp; EMIT (L.place ':=' c(Elist.array)); EMIT (L.offset ':=' Elist.place '*' width(Elist.array); }
L → id	{L.place = lookup(id.name); L.offset = null; }

Elist → Elist, E	{t := newtemp;
	m := Elist1.ndim + 1;
	EMIT (t ':=' Elist1.place '*' limit(Elist1.array, m));
	EMIT (t, ':=' t '+' E.place);
	Elist.array = Elist1.array;
	Elist.place := t;
	Elist.ndim := m;
	}

Elist → id[E	{Elist.array := lookup(id.name); Elist.place := E.place Elist.ndim := 1; }

Where:

ndim denotes the number of dimensions.
Limit (array, i) function returns the upper limit along with the dimension of array

width(array) returns the number of bytes for one element of array.

Procedures call

Procedure is an important and frequently used programming construct for a compiler. It is used to generate good code for procedure calls and returns.

Calling sequence:

The translation for a call includes a sequence of actions taken on entry and exit from each procedure. Following actions take place in a calling sequence:

- When a procedure call occurs then space is allocated for activation record.
- Evaluate the argument of the called procedure.
- Establish the environment pointers to enable the called procedure to access data in enclosing blocks.
- Save the state of the calling procedure so that it can resume execution after the call.
- Also save the return address. It is the address of the location to which the called routine must transfer after it is finished.
- Finally generate a jump to the beginning of the code for the called procedure.

Let us consider a grammar for a simple procedure call statement

1. S → call id(Elist)
2. Elist → Elist, E
3. Elist → E

A suitable transition scheme for procedure call would be:

Production Rule	**Semantic Action**
S → call id(Elist)	for each item p on QUEUE do GEN (param p) GEN (call id.PLACE)
Elist → Elist, E	append E.PLACE to the end of QUEUE
Elist → E	initialize QUEUE to contain only E.PLACE

Queue is used to store the list of parameters in the procedure call.

Declarations

When we encounter declarations, we need to lay out storage for the declared variables.
For every local name in a procedure, we create a ST(Symbol Table) entry containing:

1. The type of the name
2. How much storage the name requires

The production:

1. D → integer, id
2. D → real, id
3. D → D1, id

A suitable transition scheme for declarations would be:

Production rule	**Semantic action**
D → integer, id	ENTER (id.PLACE, integer) D.ATTR = integer
D → real, id	ENTER (id.PLACE, real) D.ATTR = real
D → D1, id	ENTER (id.PLACE, D1.ATTR) D.ATTR = D1.ATTR

ENTER is used to make the entry into symbol table and **ATTR** is used to trace the data type.

Case Statements

Switch and case statement is available in a variety of languages. The syntax of case statement is as follows:

```
1.  switch E
2.          begin
3.                  case V1: S1
4.                  case V2: S2
5.                  .
6.                  .
7.                  .
8.  case Vn-1: Sn-1
9.  default: Sn
10.         end
```

The translation scheme for this shown below:

Code to evaluate E into T

```
1.  goto TEST
2.          L1:     code for S1
3.                  goto NEXT
4.          L2:     code for S2
5.                  goto NEXT
6.                  .
7.                  .
8.                  .
9.          Ln-1:   code for Sn-1
10.                 goto NEXT
11.         Ln:     code for Sn
12. goto NEXT
13.         TEST:    if T = V1 goto L1
14.                  if T = V2goto L2
15.                  .
16.                  .
17.                  .
18.                  if T = Vn-1 goto Ln-1
19.                  goto
20. NEXT:
```

- When switch keyword is seen then a new temporary T and two new labels test and next are generated.
- When the case keyword occurs then for each case keyword, a new label Li is created and entered into the symbol table. The value of Vi of each case constant and a pointer to this symbol-table entry are placed on a stack.