UNIT-V

Syntax Directed Translation & Intermediate Code Generation

Syntax directed definition:

- In syntax directed translation, along with the grammar which associates some informal notations and these notations are called as semantic rules.
- In syntax directed translation, every non- terminal can get one or more than one attribute or sometime 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, attributes is VAL and an attributes 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.

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
- num.lexval is the attribute returned by the lexical analyzer.
- The syntax directed translation scheme is a context free grammar.
- The syntax directed translation scheme is used to evaluate the order of 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.
- The syntax directed translation scheme is useful because it enables the compiler designer to express the generation of intermediate code directly in terms of the syntactic structure the source language.

Eg. suppose output action α is associated with A→xyz

- The action α is executed whenever syntax analyzer recognizes its input.
- It bottom up parser the action is taken when xyz is reduced to A
- In top down parsing action is taken when A is expanded to xyz.
- The output action involves:
  - i. The computation of values.
  - ii. Generation of intermediate code
  - iii. Printing of errors.
- The values associated with a grammar symbol is called a translation of that symbol.
- The translation consists of various field and rules as we wish.

Eg.

E→E+E

$E.val := E^1.val+E^2.val$

- Semantic action is enclosed in braces.

Semantic Rules:

There are two notations for semantic rules:

i. Syntax directed definitions:
It is a context free grammar or in which grammar have a set of attributes.

ii. Translation schemes:
It indicates the order in which semantic rules are to be evaluated.

Implementation of syntax directed translation:

- Syntax directed translation scheme provides a method for describing an input-output mapping and that description is independent of any implementation.
- Syntax directed translation is implemented by constructing a parse tree and performing the action in a left to right depth first order.
- Syntax directed translation implementing by parse the input and produce a parse tree as a result.

Eg.

Production	**Semantic Rules**
S→E$	{print E.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}
E→I digit	{I.VAL:=10*I.VAL+LEX.VAL}
I→digit	{I.VAL:=LEXVAL}

Parsee tree for SDT:

- After written syntax directed translation scheme, next step is to convert it into a program.
- One way to implement syntax directed translation is to use extra field in stack for grammar symbols.
- These extra fields holds the values of translations.
- 1) suppose the stack.
  2) It is implemented by using arrays STATE and VAL.
  3) Each STATE is a pointer to parsing table .
  4) If the STATE symbol is E then VAL will hold the value of translation E.VAL.
  5) Top is a pointer to the current top of stack.
  6) Eg: $A \longrightarrow xyz$

Before , xyz is reduced to A the first value z is in VAL[TOP]
Second value y is in VAL[TOP+1].
Third value x is in VAL[TOP+2].

STATE	VAL
Z	Z.VAL
Y	Y.VAL
X	X.VAL
: :	: :

Fig : STACK

# Intermediate code generation:

- 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.

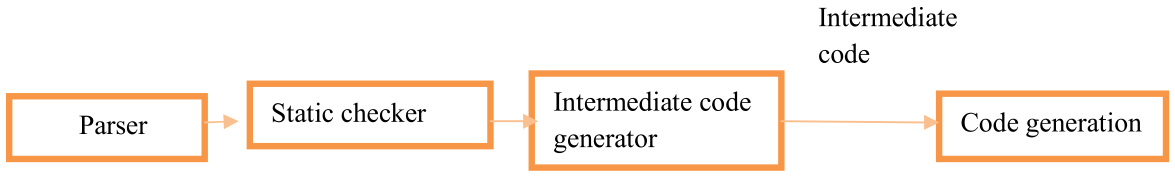


Fig. Position of intermediate code generator

- If the compiler directly translates source code into the machine code then a full native compiler is required for the each new machine.
- The 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 int two ways:

a) 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.

b) 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.

# Three address code:

- Three address codes is a type of intermediate code which is easy to generate and can be easily converted to machine code.
- It makes use of the at most three addresses and one operator to represent an expression and the value computed at each instruction is stored in temporary variable generated by compiler.
- The compiler decides the order of operation given by three address code.

General Representation:

a=b op c

where a,b and c represents operands like names, constants or compiler generated temporaries and op represents the operator.

Eg: convert the expression a* -(b+c) into three address code

t1=b+c

t2=a*-(t1)   .: - (t1) =t2

t3=a*t2

Implementation of three address code

-There are 3 representations of 3-address code namely

a)Qudraples

b)Triples

c)Syntax tree

d) postfix Notation

a) **Qudraples :**

It is structure with consists of 4 fields namely op,arg1,arg2 and result.
Op notes the operator.
Arg1 and arg2 denotes two operands and result is used to store the result of the expression.

b) **Triples :**

This representation doesn't make use of extra temporary variable to represent a single operation instead when a refrence to another triple's value is needed, a pointer to that triple is used . so it consists of only three fields namely op,arg1,arg2.

#Postfix Notation :

- Postfix notation is the useful from 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 xy*
- In postfix notation , the operator follows the operand.
- No parentheses are needed in postfix notation

  Eg : 1) (a+b)*c $\longrightarrow$ ab+c*

  2) a*(b+c) $\longrightarrow$ abc +*

  3) (a+b)*(c+d) $\longrightarrow$ ab+cd +*

  Eg : consider the postfix ab+c*
- The right hand * says that there are two arguments.
- The next to the symbol is c i.e. c is second operand of *.
- Next to c is operator +.
- + says that, it is operator between a and b.
- One language that uses a postfix intermediate language is SNOBOL.
- The SNOBOL is interpreted rather than compiled.
- The output of the SNOBOL is intermediate code itself.

# Evaluation of postfix Notation

- The postfix expression is evaluated using a stack .
- The postfix code is scanned from left to right .
- We push each operator onto the stack .
- If we found a ‘k’ operator its first argument will be k-1 position .
- The last argument will be at top .
- These values are popped and result is pushed onto the stack .

Eg: consider the postfix expression

ab+c*

suppose a,b and c have values 1,3 and 5 respectively.

To evaluate 13+5* we perform following action

1) Stack 1
2) Stack 3
3) Add the two top nest element pop them off the stack.
4) Stack 5.
5) Multiply two top most elements pop off the stack.

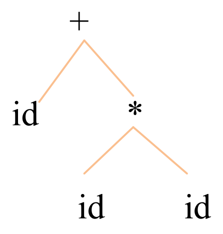
# Parse tree and syntax tree :

1) To create a parse tree it contains more details than actually needed.
2) So it is very difficult to compiler to parse the parse tree.
3) Parse tree is a useful intermediate language for source program.
4) Another variation of a parse tree is called as syntax tree.
5) In the parse tree, most of leaf nodes are single child to their parent nodes.
6) In the syntax tree, w can eliminate this extra information.
7) Syntax tree is variant of parse tree. In the syntax tree, interior nodes are operators and leaves are operands.
8) Syntax tree is usually used when represent a program in a tree structure.
9) Syntax tree having leaf node and interior node.
10) Interior node contains operator and child node contains operand.
11) In parse tree node represent the constructs and children of node represent component of construct.
12) Another variation of syntax tree is directed acyclic graph (DAG)
13) The DAG having leaf which contains operand.

The interior node contains operator but DAG has more than one interior node.

Eg : id+id*id

The syntax tree for above i/p string is :

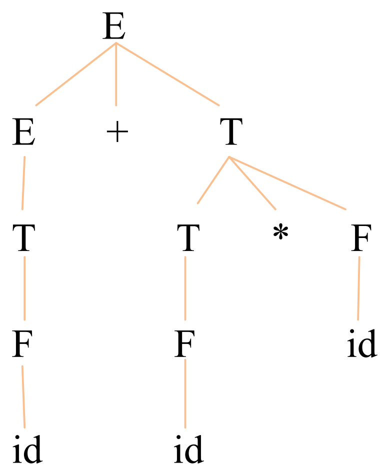


Eg : $E \rightarrow E+T/T$

$T \rightarrow T*F/F$

$F \rightarrow id$

The parse tree for above grammar is

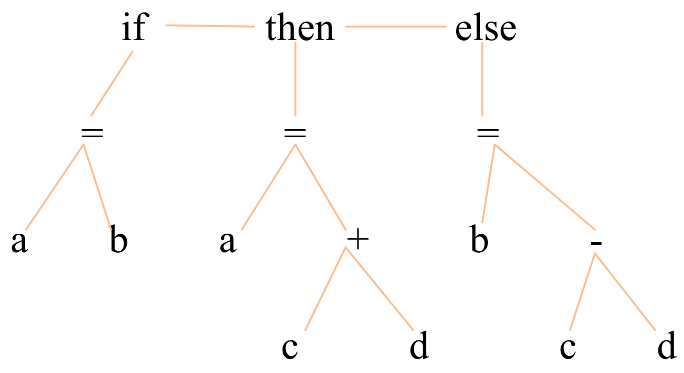


Eg : if a=b then

a=c+d else

b= c-d

The parse tree for above statement is


if
then
else
=
=
=
a
b
a
+
b
-
c
d
c
d