MODULE-5

- **Syntax Directed Translation**
- **Intermediate code generation**
- **Code generation**

Introduction

- We can associate information with a language construct by attaching attributes to the grammar symbols.
- A syntax directed definition specifies the values of attributes by associating semantic rules with the grammar productions.

Ordering the evaluation of attributes

If dependency graph has an edge from M to N then M must be evaluated before the attribute of N

Thus the only allowable orders of evaluation are those sequence of nodes N1,N2,…,Nk such that if there is an edge from Ni to Nj then i<j

Such an ordering is called a topological sortof a graph

Example!

S-Attributed definitions

An SDD is S-attributed if every attribute is synthesized

We can have a post-order traversal of parse-tree to evaluate attributes in S-attributed definitions

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

S-Attributed definitions can be implemented during bottom-up parsing without the need to explicitly create parse trees

L-Attributed definitions

- A SDD is L-Attributed if the edges in dependency graph goes from Left to Right but not from Right to Left.
- More precisely, each attribute must be either
    - Synthesized
    - Inherited, but if there us a production A->X1X2…Xn and there is an inherited attribute Xi.a computed by a rule associated with this production, then the rule may only use:
        - Inherited attributes associated with the head A
        - Either inherited or synthesized attributes associated with the occurrences of symbols X1,X2,…,Xi-1 located to the left of Xi
        - Inherited or synthesized attributes associated with this occurrence of Xi itself, but in such a way that there is no cycle in the graph

Application of Syntax Directed Translation

- Construction of syntax trees
    - Leaf nodes: Leaf(op,val)
    - Interior node: Node(op,c1,c2,…,ck)

Example:

Production

E -> E1 + T

E -> E1 - T

E -> T

T -> (E)

T -> id

T -> num

Semantic RULE

E.node=new node(‘+’, E1.node,T.node)

E.node=new node(‘-’, E1.node,T.node)

E.node = T.node

T.node = E.node

T.node = new Leaf(id,id.entry)

T.node = new Leaf(num,num.val)

Syntax tree for L-attributed definition

Syntax directed translation schemes

An SDT is a Context Free grammar with program fragments embedded within production bodies

Those program fragments are called semantic actions

They can appear at any position within production body

Any SDT can be implemented by first building a parse tree and then performing the actions in a left-to-right depth first order

Typically SDT's are implemented during parsing without building a parse tree .

Postfix translation schemes

Simplest SDDs are those that we can parse the grammar bottom-up and the SDD is s-attributed

For such cases we can construct SDT where 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 postfix SDT's

Parse-Stack implementation of postfix SDT's

In a shift-reduce parser we can easily implement semantic action using the parser stack

For each nonterminal (or state) on the stack we can associate a record holding its attributes

Then in a reduction step we can execute the semantic action at the end of a production to evaluate the attribute(s) of the non-terminal at the leftside of the production

And put the value on the stack in replace of the rightside of production

EXAMPLE

```
L -> E n        {print(stack[top-1].val);
                                top=top-1;}
E -> E1 + T     {stack[top-2].val=stack[top-2].val+stack.val;
                                top=top-2;}
E -> T
T -> T1 * F     {stack[top-2].val=stack[top-2].val+stack.val;
                                top=top-2;}
T -> F
F -> (E)        {stack[top-2].val=stack[top-1].val
                        top=top-2;}
F -> digit
```

Intermediate Code Generation

- Intermediate code is the interface between front end and back end in a compiler
- Ideally the details of source language are confined to the front end and the details of target machines to the back end (a m*n model)
- In this chapter we study intermediate representations, static type checking and intermediate code generation.

Variants of syntax trees

- It is sometimes beneficial to crate a DAG instead of tree for Expressions.
- This way we can easily show the common sub-expressions and then use that knowledge during code generation
- Example: a+a*(b-c)+(b-c)*d

SDD for creating DAG'sSDD for creating DAG's

Value-number method for constructing DAG's

- Algorithm
    - Search the array for a node M with label op, left child l and right child r
    - If there is such a node, return the value number M
    - If not create in the array a new node N with label op, left child l, and right child r and return its value
- We may use a hash table

Three address code

- **In a three address code there is at most one operator at the right side of an instruction**

Example:

Data structures for three address codes

- Quadruples
    - Has four fields: op, arg1, arg2 and result
- Triples
    - Temporaries are not used and instead references to instructions are made
- Indirect triples
    - In addition to triples we use a list of pointers to triples.

Type Expressions

Example: int[2][3]

array(2,array(3,integer))

A basic type is a type expression

A type name is a type expression

A type expression can be formed by applying the array type constructor to a number and a type expression.

A record is a data structure with named field

A type expression can be formed by using the type constructor g for function types

If s and t are type expressions, then their Cartesian product s*t is a type expression

Type expressions may contain variables whose values are type expressions.

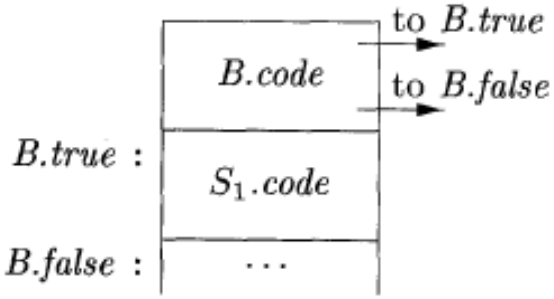
Short-Circuit Code

```
if ( x < 100 || x > 200 && x != y ) x = 0;
```

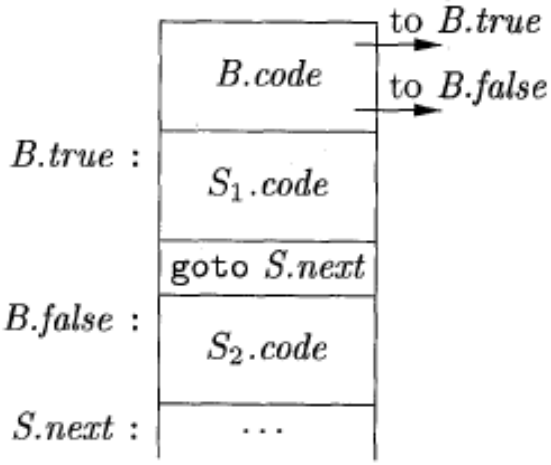
```
        if x < 100 goto L2
        ifFalse x > 200 goto L1
        ifFalse x != y goto L1
L2 :    x = 0
L1 :
```

Flow-of-Control Statements

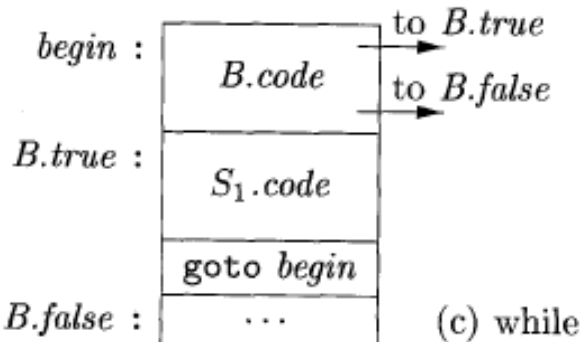
$S \rightarrow$ **if** (B) S_1
$S \rightarrow$ **if** (B) S_1 **else** S_2
$S \rightarrow$ **while** (B) S_1



(a) if



(b) if-else



(c) while