Unit 4 : Syntax-Directed Translation & intermediate code generation

Introduction:

- In case of Lexical phase and syntax phase of compilation, we have a ***Regular expression*** as a notation to describe the ***Lexical structure*** of the programming language and a ***Grammar*** as a notation to describe the ***Syntactic structure*** of the programming language.
- Similar notation is expected by the remaining phases of compiler. Unfortunately it is not there.
- However we have a ***notational framework*** which is an extension of ***context free grammar***. This notational framework is called as ***syntax directed translation scheme***. According to this we have a ***Set of Attributes*** for each grammar symbol and ***Semantic rules*** are attached with each production. when these semantic rules are called at appropriate time they evaluate the values for the attributes and may generate the required intermediate code.

Notations for associating semantic rules with productions:

Basically there are two notational frameworks for associating semantic rules with productions namely

Syntax-Directed Definitions (SDDs)

Syntax –Directed Translation Schemes (SDTs) or simply Translation schemes

1. **Syntax-Directed Definitions(SDDs):**
   - give high-level specifications for translations
   - hide many implementation details such as order of evaluation of semantic actions.
   - We associate a production rule with a set of semantic actions, and we do not say when they will be evaluated.
2. **Syntax –Directed Translation Schemes(SDTs):**
   - These are also high level specifications for translations
   - Implementation details are not hidden. In other words, translation schemes gives detailed information about implementation.
   - indicate the order of evaluation of semantic actions associated with a production rule.

- ***What are semantic rules?***

  Semantic rules are nothing but some action routines, which when called at appropriate time may evaluate the values for the attributes of the grammar symbol and generate the intermediate code for the construct. In addition to that they also performs other tasks namely,

  - may put information into the symbol table
  - may perform type checking
  - may issue error messages
  - may perform some other activities
  - in fact, they may perform almost any activities.

- ***What are Attributes of Grammar symbol. ?***

  Attribute of grammar of grammar symbol is a term that represents value or may hold almost any thing i.e

  a string, a number, a pointer to memory location, a Boolean value, a complex record .

  These values are defined by the semantic actions associated with the production

- ***What are Types of attributes ?***

  Basically there are two types of attributes for the grammar symbols namely **synthesized attribute** and **Inherited attributes.**

  The ***synthesized attribute for a Non terminal at a parse tree node N*** is the values of attribute that are computed from the ***values of the attribute at the children*** of that node in the parse tree.

  The ***Inherited attributes for a Non terminal at a parse node N is*** the values of the attribute that are computed from the ***values of the attribute at their siblings and/or parents*** of that node in the parse tree.

Form of Syntax-Directed Definitions:

A syntax-directed definition is a generalization of a context-free grammar in which:

- Each grammar symbol is associated with a set of attributes.
- This set of attributes for a grammar symbol is partitioned into two subsets called
  - **synthesized** and
  - **inherited** attributes of that grammar symbol.
- Each production rule is associated with a set of semantic rules.

In a syntax-directed definition, each production **A→α** is associated with a set of semantic rules of the form: $b=f(c_1,c_2,\ldots,c_n)$

where f is a function and ***b*** can be one of the followings:

b is a synthesized attribute of **A *and c1,c2,…,cn*** are attributes of the grammar symbols in the production (**A→α**).

OR

b is an inherited attribute one of the grammar symbols in **α** (on the right side of the production), and **c1,c2,…,cn** are attributes of the grammar symbols in the production (**A→α**).

Annotated Parse Tree, Annotating a Parse tree and Dependency Graph:

- A parse tree showing the values of attributes at each node is called an **annotated parse tree**.
- The process of computing the attributes values at the nodes is called **annotating** (or **decorating**) of the parse tree.Of course, the order of these computations depends on the dependency graph induced by the semantic rules.

- A dependency graph is a useful tool that are used to determine the order of computation of attributes in parse tree. Annotated parse tree shows the values of attributes at each node and Dependency graph helps us determine how these values can be computed.

Dependency graph:

- For each parse tree, the parse tree has a node for each attribute associated with that node
- If a semantic rule defines the value of synthesized attribute **A.b** in terms of the value of **X.c** then the dependency graph has an edge from **X.c** to **A.b**
- If a semantic rule defines 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**

Attribute Grammar:

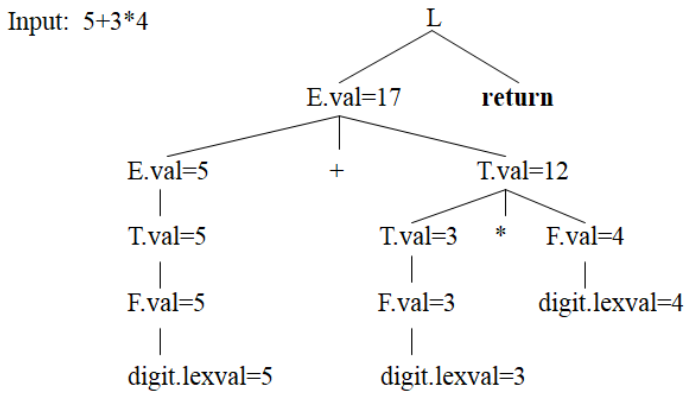
- So, a semantic rule $b=f(c_1,c_2,\ldots,c_n)$ indicates that the attribute b *depends* o*n* attributes $c_1,c_2,\ldots,c_n$.
- In a **syntax-directed definition**, a semantic rule may just evaluate a value of an attribute or it may have some side effects such as printing values.
- An **attribute grammar** is a syntax-directed definition in which the functions in the semantic rules cannot have side effects (they can only evaluate values of attributes).

Syntax-Directed Definition of a Simple Desk Calculator – Example:

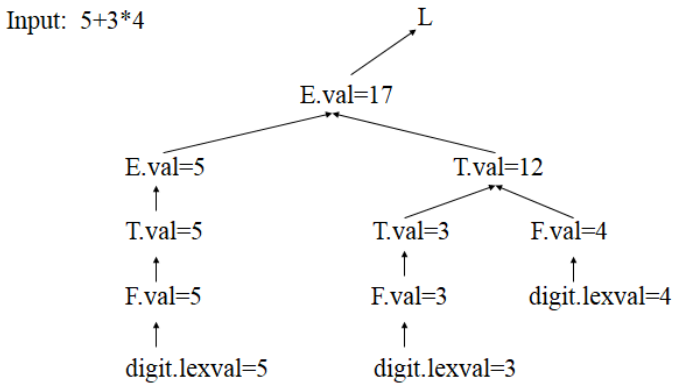
Production	**Semantic Rules**
L → E **return**	print(E.val)
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
E → T	E.val = T.val
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
T → F	T.val = F.val
F → (E)	F.val = E.val
F → **digit**	F.val = **digit**.lexval

- Symbols E, T, and F are associated with a synthesized attribute *val*.
- The token **digit** has a synthesized attribute *lexval* (it is assumed that it is evaluated by the lexical analyzer).

Annotated Parse Tree – Example:



Dependency Graph:



Syntax-Directed Definition – With Inherited Attributes:

- Declaration statement
  - Input statement and Grammar specifications
  - Writing Semantic actions

    * Role of Semantic actions

      - For each identifier the semantic actions must enter type information into the symbol table entry for the identifier.
      - Defining Attributes for grammar symbols

    *Attaching semantic actions

    *Execution of semantic actions and Trace

Syntax-Directed Definition – Inherited Attributes:

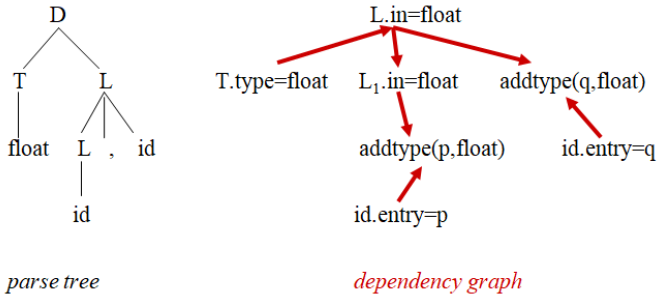
Production	**Semantic Rules**
1. D → T L	L.in = T.type
2. T → **int**	T.type = integer
3. T → **float**	T.type = float
4. L → L_1, **id**	L_1.in = L.in, addtype(**id**.entry,L.in)

5. L → **id** addtype(**id**.entry,L.in)

- Symbol T is associated with a synthesized attribute *type*.
- Symbol L is associated with an inherited attribute *in.*
- From production 1: **D represents a declaration & consists of a type T followed by a list L of identifiers.**
  From production 2 & 3: **evaluate T.type**
  From production 4 & 5: **L1.in is computed at a node by copying the value of L.in from the parent node.**
  The addType has two arguments id.entry a lexical value that points to a symbol table object and L.in the type being assigned to every identifier on the list

A Dependency Graph – Inherited Attributes:

Input: `float p, q`



parse tree *dependency graph*

SDD for constructing Syntax Trees:

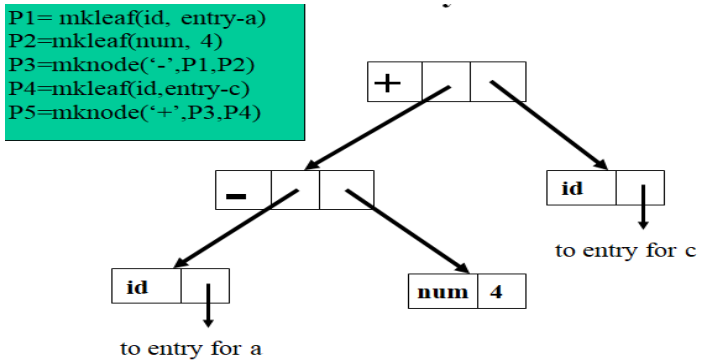
- Each node in a syntax tree is implemented as a record with several fields. One field identifies the operator and remaining fields contain pointers to the nodes for the operands.
- We assume three functions to create the nodes for the syntax trees for expressions with binary operators.
  - ***mknode(op,left,right)*** – This creates an operator node with label 'op' and two fields containing pointers to left and right child
  - b)***mkleaf(id, id.entry)*** - This creates a leaf node with label id and a field containing entry, a pointer to the symbol entry to the identifier.
  - ***mkleaf(digit, digit.val)*** - This creates a leaf node with label num and a field containing val, the value of the number

Production	**Semantic Rule**
E → E_1 + T	**E.*nptr* = *mknode*("+",E_1.*nptr* ,T.*nptr*)**
E → E_1 - T	**E.*nptr* = *mknode*("-",E_1.*nptr* ,T.*nptr*)**
E → T	**E.*nptr* = T.*nptr***
T → T * F	**T.*nptr* = *mknode*("*",T_1.*nptr* ,F.*nptr*)**
T → F	**T.*nptr* = F.*nptr***
F → (E)	**F.*nptr* = E.*nptr***

F → id **F.*nptr* = *mkleaf*(id, id.*entry*)**

F → digit **F.*nptr* = *mkleaf*(num, digit.*val*)**

Syntax Tree for a-4+c:



Directed Acyclic Graphs (DAG):

DAG is an important intermediate representation used by the compiler and it has the following properties

1. Leaves correspond to atomic operands
2. Interior nodes correspond to operators
3. A node N in a DAG can have more than one parent if N represents a common sub-expression

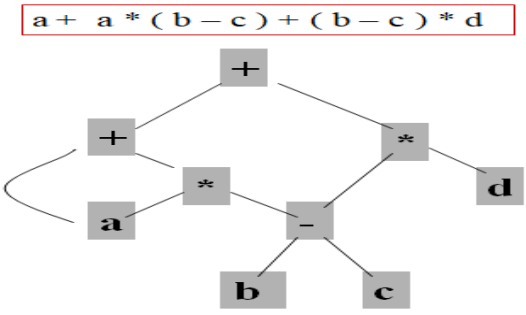
Advantages:

1. Represents expressions more succinctly
2. Identifies the common sub-expressions
3. Gives the compiler more clues for generation of efficient code

Constructing a DAG:

- A syntax directed definition is used to construct a DAG
- The steps are similar to the construction of syntax trees
- But before creating a new node we need to check whether an identical node already exists
- If such a node exists then pointer to an existing node is returned
  else a new node is created and pointer to a newly created node is returned.

Directed Acyclic Graphs for Expressions:



Draw DAG for the following:

1. A+b+(a+b)
2. A+b+a+b
3. (a*b)+(c+d)*(a*b) + b
4. ((x+y)-((x+y)*(x-y))) + ((x+y)*(x-y))
5. A+a+((a+a+a+(a+a+a+a))

Syntax Directed Definition for Infix to postfix Expression:

- The ordinary way of writing the sum of a and b is placing the operator in the middle i.e a+b.
- The postfix notation for the same expression places the operator at the right end as ab+.
- If E1=a and E2= b are two postfix notation and '+' is binary operator then apply + on E1 and E2 to write postfix notation is ab+
- In general, if E1 and E2 are any two postfix expression and Θ is any binary operator, the result of applying the Θ to the values denoted by E1 and E2 is indicated in postfix notation by E1E2Θ.
- In order to write SDD that generates Intermediate code as Postfix expression, the string valued attribute can be assumed and is denoted as E.code where E is grammar symbol
- Consider the production $E \rightarrow E_1 + T$, the grammar symbols E, and T assumed to have string valued attribute E.code and T.code.
- The attribute E.code of head of the production is defined in terms of attributes E_1.code of E_1 and T.code of T which are the grammar symbols of the RHS of the production. This is nothing but the E.code contains the concatenation of E_1.code and T.code followed by the corresponding operator '+'.
- The semantic action for the production is written as follows :
  Production Semantic Actions.

  $E \rightarrow E_1 + T$ E.*code*= E1.code || T.code || *'+'*

Syntax Directed Definition for Infix to postfix Expression:

PRODUCTION	**SEMANTIC RULE**
$E \rightarrow E_1 + T$	E.*code*= E1.code || T.code || *'+'*
$E \rightarrow T$	E.code=T.code
$T \rightarrow T_1 * F$	T.*code*= T1.code || F.code || *'*'*
$T \rightarrow F$	T.code = F.code
$F \rightarrow (E)$	F.code = E.code
$F \rightarrow id$	F.*code* = getname(id.place)
$F \rightarrow num$	F.*Code* = num.lexval

Sub-classes of SDD:

1. Syntax-directed definitions are used to specify syntax-directed translations.

2. We would like to evaluate the semantic rules during parsing (i.e. in a single pass, we will parse and we will also evaluate semantic rules during the parsing).
3. We will look at two sub-classes of the syntax-directed definitions:
    a. **S-Attributed Definitions**: only synthesized attributes used in the syntax-directed definitions.
    b. **L-Attributed Definitions**: in addition to synthesized attributes, we may also use inherited attributes in a restricted fashion.
4. To implement S-Attributed Definitions and L-Attributed Definitions we can evaluate semantic rules in a single pass during the parsing.
5. Implementations of S-attributed Definitions are a little bit easier than implementations of L-Attributed Definitions

Syntax Directed Translation Schemes:

- SDT's are complementary notation to Syntax Directed Definitions
- All SDD's discussed can be implemented using SDTs
- Definition:

  A ***syntax directed translation scheme (SDTs)*** is a context free grammar with program fragments embedded within production bodies. These program fragments are called semantic actions and can appear at any position within a production bodies.
- By conventions semantic actions are enclosed by curly braces.
- Any SDTs can be implemented by first building a parse tree and then performing the actions in post order traversal.
- We shall discuss SDTs implementations on LR parser and SDD is S-attributed.

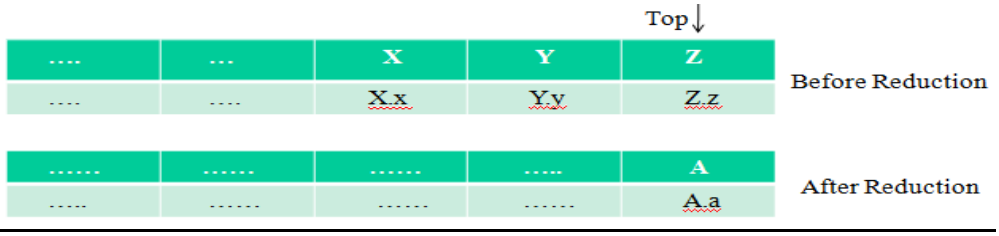
Parser-Stack Implementation of Postfix SDT's:

- SDTs with all semantic actions at the right ends of the production bodies is called ***postfix SDTs***
- Postfix SDT's can be implemented during LR parsing by executing the actions when reductions occur.
- We can construct an SDTs in which actions are placed at the end of the production and are executed along with the reduction of the body to the head of the production.
- 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 in records on the stack itself

Bottom-Up Evaluation of S-Attributed Definitions:

- The parser stack is implemented as a record with two field, one field for a grammar symbol(or parser state) and another for an attribute. When an entry of the parser stack holds a grammar symbol X (terminal or non-terminal), the corresponding entry in the stack will hold the synthesized attribute(s) of the symbol X
- For example : $A \rightarrow XYZ$ A.a=f(X.x,Y.y,Z.z) where all attributes are synthesized.
    - The three grammar symbols XYZ are on top of the stack and they are reduced according to the production $A \rightarrow XYZ$. Here X.x,Y.y and Z.z are the synthesized attributes of X Y and Z. and X.x is at top-2, Y.y is at top-1 and Z.x is at top of the stack.

- We evaluate the values of the attributes during reductions and is as follows.



Postfix SDT implementing the desk calculator:

L->E n {print (E.val);}

E->E1+T {E.val=E1.val + T.val;}

E-> T {E.val=T.val;}

T->T1*F {T.val=T1.val*F.val;}

T->F {T.val=F.val;}

F->(E) {F.val=E.val;}

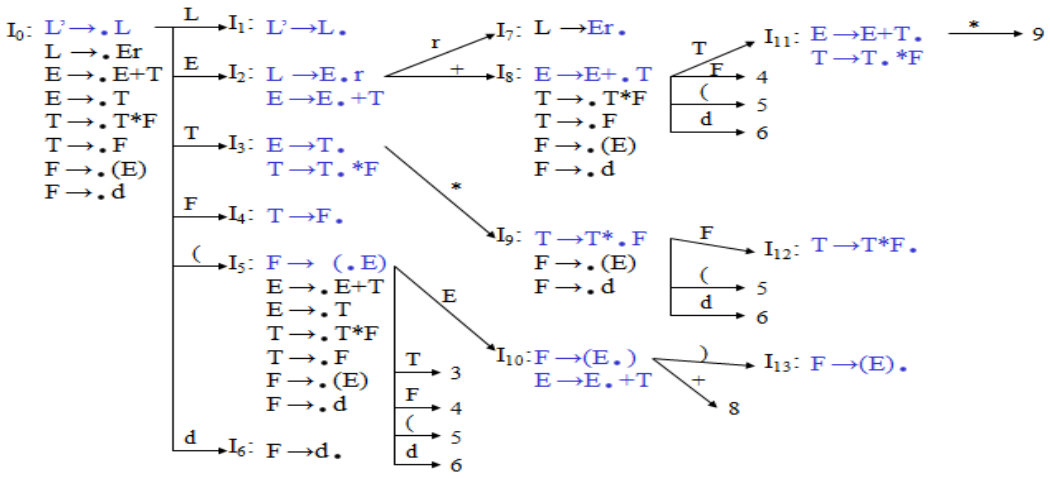
F->digit {F.val=digit.lexval;}

Postfix SDT implementing the desk calculator on bottom-Up Eval. Of S-Attributed Definitions:

Production	**Semantic Actions**
L → E **return**	{ print(stack[top-1].val) }
$E \rightarrow E_1 + T$	{ stack[top-2].val = stack[top-2].val+ stack[top].val top=top-r + 1 }
E → T	
$T \rightarrow T_1 * F$	{ stack[top-2].val = stack[top-2].val * stack[top].val top=top-r + 1 }
T → F	
F → (E)	{ stack[top-2].val = stack[top-1].val top=top-r + 1 }
F → **digit**	

- At each shift of **digit**, we also push **digit.lexval** into *val-stack*.
- At all other shifts, we do not put anything into *val-stack* because other terminals do not have attributes (but we increment the stack pointer for *val-stack*).

Canonical LR(0) Collection for The Grammar:



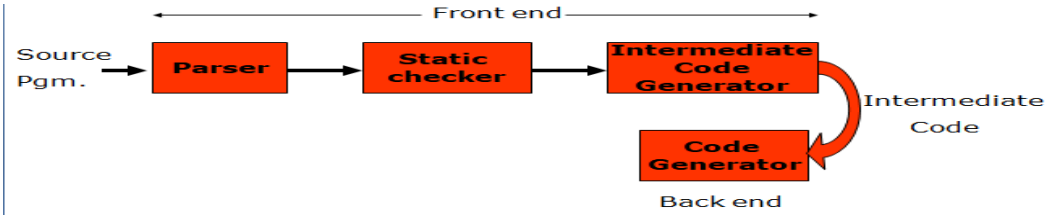
Bottom-Up Evaluation – Example:

- At each shift of **digit**, we also push **digit.lexval** into *val-stack*.

stack	val-stack	input	action	semantic Actions
0		5+3*4r	s6	d.lexval(5) into val-stack
0d6	5	+3*4r	F→d	F.val=d.lexval – do nothing
0F4	5	+3*4r	T→F	T.val=F.val – do nothing
0T3	5	+3*4r	E→T	E.val=T.val – do nothing
0E2	5	+3*4r	s8	push empty slot into val-stack
0E2+8	5-	3*4r	s6	d.lexval(3) into val-stack
0E2+8d6	5-3	*4r	F→d	F.val=d.lexval – do nothing
0E2+8F4	5-3	*4r	T→F	T.val=F.val – do nothing
0E2+8T11	5-3	*4r	s9	push empty slot into val-stack
0E2+8T11*9	5-3-	4r	s6	d.lexval(4) into val-stack
0E2+8T11*9d6	5-3-4	r	F→d	F.val=d.lexval – do nothing
0E2+8T11*9F12	5-3-4	r	T→T*F	T.val=T_1.val*F.val
0E2+8T11	5-12	r	E→E+T	E.val=E_1.val*T.val
0E2	17	r	s7	push empty slot into val-stack
0E2r7	17-	$	L→Er	print(17), pop empty slot from val-stack
0L1	17	$	acc	

Intermediate code generation:

- Intermediate code is a form of representation of source statement that is very close to machine instructions and independent of machine code
- Analysis-synthesis model:
  - Front end analyses a source code and creates an intermediate representation
  - From this intermediate representation the back end generates the object code
  - The front end is program dependent and the back end is machine dependent
- We assume that a compiler front end is organized as in the figure shown below
- Here parsing, static checking, and intermediate-code generation are done sequentially.



Benefits of Intermediate codes:

1. Intermediate code is closer to the target machine than the source language, and hence easier to generate the code form
2. Unlike machine language, intermediate code is machine independent. This makes it easier to retarget compiler.
3. It allows a variety of machine independent optimizations to be performed.
4. Typically, intermediate code generation can be implemented via SDTs and thus can be folded into parsing and Type checking.

Different kinds of Intermediate codes:

There are Two kinds of intermediate forms namely

1. High level Intermediate representation

   These representations expresses high level structure of the program which is closer to source program and can be generated easily form the input program

   Example : Abstract syntax tree and DAG

2. Low level Intermediate representation

   These representations expresses low level structure of the program which is closer to M/c program and generations may involve some work

   Example : Three address code, P-code, Diana and Byte code

Directed Acyclic Graphs (DAG):

- Leaves correspond to atomic operands
- Interior nodes correspond to operators
- A node N in a DAG can have more than one parent if N represents a common subexpression

Advantages:

- Represents expressions more succinctly
- Gives the compiler more clues for generation of efficient code

Constructing a DAG:

- A syntax directed definition is used to construct a DAG
- The steps are similar to the construction of syntax trees
- But before creating a new node we need to check whether an identical node already exists
- If such a node exists the existing node is returned
  else a new node is created.

The value-number method:

- Nodes of a syntax tree or DAG can be stored as an array of records. Each row of the array represents a node
- In each record the first field represents an operation code
- For leaves one additional field holds the lexical value
- For interior nodes there are two additional fields for left and right children
- We refer to each node with integer index of the array called the value number

Algorithm for value-number method:

//Suppose that nodes are stored in an array and each node is referred to by its value number.

- **Input**: label op,node l and node r
- **Output**:the value of node in the array with signature <op, l, r>
- **Method**:search the array for the node with label op,left child l & right child r.If found return its value number.If not found,we create in the array a new node with label op left child l and right child r & return its value number

DAG Construction
i = i + 10

Construct the DAG for the expression

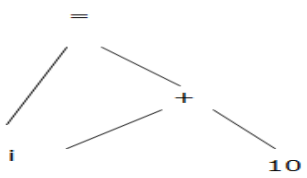
a+b+(a+b)

a+b+a+b

((x+y)-((x+y)*(x-y))) + ((x+y)*(x-y))

a+a+((a+a+a+(a+a+a+a))

DAG Construction for i = i + 10:



1	id	To entry for I	
2	num	10	
3	+	1	2
4	=	1	3

Hash table and buckets:

- Using hash tables are more efficient.
- Hash table is a array of 'buckets'.
- The index of the bucket is computed using a hash function h for a signature <op,l,r>.
- The buckets can be implemented as linked lists.An array of pointers indexed by the hash value points to first nodes of the buckets.Thus the node <op,l,r> can be found on the list whose header index is given by h(op,l,r).

Three-address code:

- In three-address code,there is at most one operator on the right side of an instuction

- If more than one operator is to be used then they are simplified
  - Eg.
  - X+y*z can be written as
  - T1=y*z
  - T2=x+T1
- Three address code is built from two concepts: addresses and one operational instructions

Addresses:

Address are of three types

- A name: source program names can appear as addresses in three adderss code
- A constant: compiler must be able to deal with different types of constants
- A compiler generated temporary are used as addresses

Three address instructions:

- Assignment instructions: x=y op z;
- Assignments of the form: x=op y;
- Copy instructions: x=u;
- An unconditional jump: goto L
- Conditional jumps(1): if x goto L or iffalse x goto L
- Conditional jumps(2): if x relop y goto L
- For procedure calls and returns : param x for parameters; call p,n and y=call p,n for procedure calls and function calls; return y where y is return value (Optional)
- Indexed copy instructions : x=y[i] and x[i]=y
- Address and pointer assignments: x=&y, x=*y,*x=y

Examples to solve:

1. X=a*(b-c)+(b-c)*d
2. A=b*-c+b*-c
3. If (x>y) then z=x+y;

   Else z=x-y;
4. While (x<y) Do

   begin

   if c>d x=x+y

   else x=x-y

   End
6. Do i=i+1; while(a[i]<v);
7. If (x<100 || x >200 && x !=y) x=0;
8. (a>b) && ((b>c) || a>c))

Quadruples:

Quadruples are used to implement the three address instructions in compilers. They have

four fields: op,arg1,arg2 & result. Exceptions are:

- Instructions with unary operators Eg x= -y(do not use arg2)
- Conditional & unconditional jumps put the target label in result.
- Operators like param [used pass the parameter] use neither arg2 nor result

Quadruple representation for b*-c+b*-c:

position	Op	Arg1	Arg2	result
1	minus	c		t1
2	*	b	t1	t2
3	minus	c		t3
4	*	b	t3	t4
5	+	t2	t4	t5
6	=	t3		a
7				

Triples:

- They are also used in the implemention of three adress instructions but use only three fields. The result field is missing here
- Using the triples we refer to the result of an operation by its position rather than by a temporary name.
- When instructions are moved around we need to change all references to that result

Indirect Triples:

- They consist of listing of pointers to tripples.
- Here we can move an instruction by reordering the instruction list without affecting the tripples themselves.

Triple representation for b*-c + b*-c:

position	op	Arg1	arg2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)

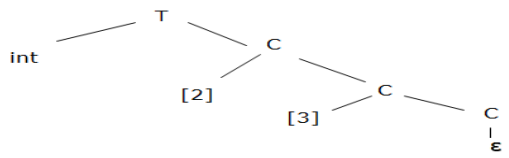
Static single-assignment form:

- SSA is an intermediate repersentation that facilitates certain code optimisations.
- All assignments are to variables with distinct names
  - p1 = a+ b
  - q1 = p1-c
  - p2= q1*d
  - p3= e-p2
  - q2= p3+q1
  - If (flag) x = -1 ;else x = 1;
  - If (flag) x1 = -1 ;else x2 = 1;
  - x3 =Ø(x1,x2);

Types and Declarations:
ex : int[2][3]

D →T id ; | D | ε
T → B C | record ' { ' D ' } '
B → int | float
C → ε | [num] C

Example:



Storage layout:

Computing types and widths:

T →B {t= B.type ; w = B.width}

C

B→ int {B.type = integer ; B.width = 4}

B→ float {B.type = float ; B.width = 8}

C → ε {C.type = t ; C.width =w}

C →[num]C1 {array(num.value,C1.type);

C.width = num.value*C1.width}

Translation of an assignment statement:

S→id = E, E→ E + E | E*E | (E) | -E | id

1. 'S' stands for assignment statement made up of two operators and all the operands denote primitive data type namely integer.
2. Intermediate code that is to be generated is sequence of Three address code statements.
3. Each Grammar symbol has a synthesize attribute. Hence synthesize attribute S.code represents a sequence of three codes for the assignment S.

The Nonterminal E has two attributes namely,

1. E.addr – denotes the address that hold the value of E. Here the address can be name or compiler generated temporary
2. E.code - denotes the sequence of three address codes evaluating E.

S→id = E If this production is used for reduction then we must have sequence of TAC (E.code) to evaluate E into some temporary and then semantic rules will generate the TAC X=E.addrs where x is the name corresponds to id and finally it has to concatenate the sequence of TAC for E.code with the TAC generated which will be S.code.

* S→id = E S.code =E.code || gen(top.get(id.lexme) '=' E.addr)

*E → E1 + E2

In this case we must have sequence of TAC (E1.code and E2.code) to evaluate E1 and E2 into some temp(E1.addr and E2.addr).

Semantic Rule must generate the new TAC E.addr=E1.addr + E2.addr and concatenate with E1.code and E2.code which will be the TAC for E

*E → E1 + E2

E.addr = new Temp()

E.code = E1.code || E2.code || Gen(E.addr '=' E1.addr '+' E2.addr)

*E→ID

In this case the expression is a single identifier say x and x itself holds the value of the expression. The semantic rule for this production must define E.addr to point to the symbol table entry for some instance of ID.

E→ID E.addr = top.get(id.lexme)

E.code =" "

Top denote the current symbol table and top.get retrieves entry form the symbol table

Translation of expressions:

Production	Semantic rules
S→id = E	S.CODE = E.CODE|| GEN(TOP.GET(ID.LEXEME '=' E.ADDR))
E → E1 + E2	E.ADDR = NEW TEMP() E.CODE = E1.CODE || E2.CODE || GEN(E.ADDR'=' E1.ADDR'+'E2.ADDR)
|-E1	E.ADDR = NEW TEMP() E.CODE = E1.CODE || GEN(E.ADDR'=' 'MINUS' E1.ADDR)
|(E1)	E.ADDR = E1.ADDR E.CODE = E1.CODE
|ID	E.ADDR = TOP.GET(ID.LEXEME) E.CODE = ""

Translation of mixed type operation in an Expression:

1. Expression may involve different types of variables and constants, so the compiler must either reject the mixed type operations or generate appropriate type conversion statements.
2. Let us consider the assignment statement with two types namely float and integer
3. For example: x= y + i * j

assume x and y have type float and i and j have type integer

Sequence of three address code could be generated as follows.

t1= i int* j

t2 = inttofloat t1

t3 = y real+ t2

x= t3

In addition to two attributes E.code and E.addr of grammar symbol E we have another attribute called E.type that represents type of operation

The semantic rules must check the type and generate the appropriate Three address code.

E → E1 + E2

```
E.ADDR = NEW TEMP();
if E1.type = integer and E2.type = integer then
Begin
E.CODE = E1.CODE || E2.CODE || GEN(E.ADDR '='
        E1.ADDR  int '+' E2.ADDR)
E.type = integer
end
else if E1.type = float and E2.type = float then
Begin
E.CODE = E1.CODE || E2.CODE || GEN(E.ADDR '='
        E1.ADDR  float '+' E2.ADDR)
E.type = float
end
 else if E1.type = integer and E2.type = float then
Begin
 u=newtemp();
E.CODE = E1.CODE || E2.CODE || u '=' inttofloat  E1.addr ||
        GEN(E.ADDR '='  E1.ADDR  float '+' E2.ADDR)
E.type = float
end
else if E1.type = float and E2.type = integer then
Begin
 u=newtemp();
E.CODE = E1.CODE || E2.CODE || u'='inttofloat E2.addr
GEN(E.ADDR '='  E1.ADDR  real '+' E2.ADDR)
E.type = float
end
```

Else E.type = type_error

Switch Statements:

- There is a selector expression which needs to be evaluated followed by a set of values that it can take.
- The expression is evaluated and depending on the value generated particular set of statements are executed
- There is always a set of default statements which is executed if no other value matches the expression

Incremental Translation:

Code attributes are usually long strings and hence are generated incrementally

Consider:

production : E -> E1 + E2

semantic rule : {E.addr=new temp()

gen(E.addr '=' E1.addr '+' E2.addr)}

Here,

gen() creates add instruction and appends it to previously generated instructions that compute E1 into E1.addr and E2 into E2.addr

Array Referrences:

Usually array elements are numbered from 0 to n-1

If width of each element is w and base is the relative address of the allocated storage,

'i'th element begins @ locn.

base + i*w

In 't' dimensions address of $a[i_1][i_2]\ldots[i_t]$ is

base + $i_1*w_1 + i_2 * w_2 + \ldots\ldots\ldots\ldots + i_t * w_t$

where w_j is the width in 'j'th dimension

This is implemented by a corresponding production/semantics

Flow of control statements:

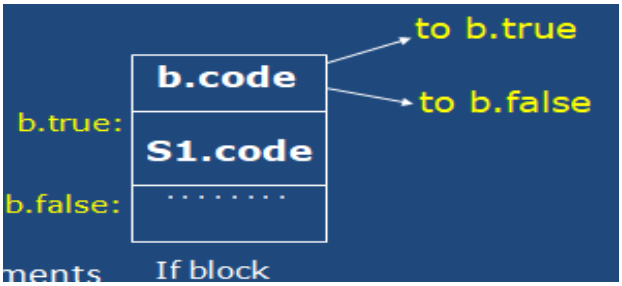
Consider the following statements:

S->if (b) s1

where s represents statements and b represents Boolean expressions.

The translation of this statement consists of b.code followed by s1.code as shown.

Based on the values of b, there are jumps within b.code. Similar are the other flow control statements.



With boolean expression B we associate two label (attributes) namely

B.true : The label to which control flows if B is true.

B.false : The label to which control flows if B is false.

we also have the following attributes

B.code : sequence of three address codes evaluating

Boolean expression.

With statement S S1, S2 (any statement) we associate label (attributes) and string attribute namely

S.next : The label to Three address code statement that follows three address code statement for S S1.next and S2.next : The label to three address code statement that follows three address

code statement for S1 and S2

S.code : sequence of three address codes of statement S. S1.code and S2.code : sequence of three address codes evaluating S1 and S2

Semantic rule for boolean expression:

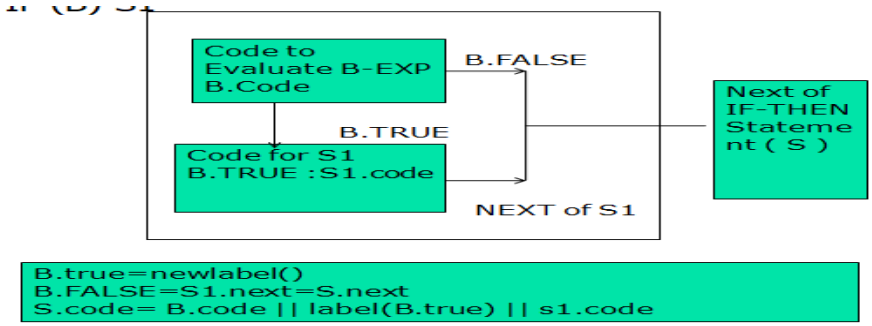
B → E1 rel E2	B.code = E1.code || E2.CODE || GEN('if' E1.addr rel.op E2.addr 'goto' B.true || GEN ('goto' B.false)
B → True	B.code = GEN ('goto' B.True)
B → False	B.code = GEN ('goto' B.false)

Example : if a<b goto B.true
goto B.false

Newlabel() – It creates a newlabel each time it is called

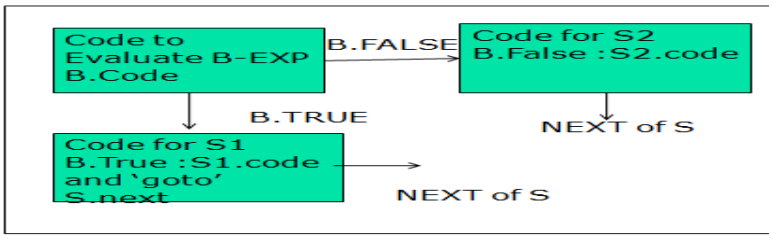
Label(L)- It attaches label L to the next three address code statement

S→IF (B) S1



B.true=newlabel()
B.FALSE=S1.next=S.next
S.code= B.code || label(B.true) || s1.code

If (B) S1 else S2



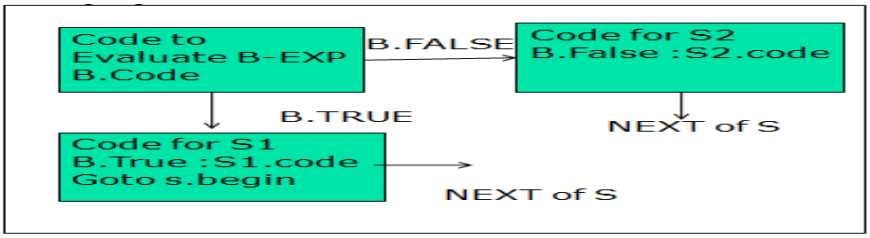
B.true=newlabel()

B.False=newlabel()

S2.next=S1.next=S.next

S.code= B.code || label(B.true) || s1.code || Gen('goto' s.next) || label(B.False) || s2.code

S→ while (b) Do s1



S->If (b) s1

b.code	
b.true:	s1.code
s.next:	.....

If (b) then s1 else s2

b.code	
b.true:	s1.code
	goto s.next
b.false:	s2.code
s.next:	.....

S-> while (b) Do s1

s.begin :	b.code
b.true :	s1.code
	goto s.begin
b.false:	.....

The semantic rule for translating if control statement must first contain sequence of TAC's of Boolean expression(b.code) followed by sequence TAC's of statement S1 with label b.true for the first statement

b.true = newlabel

b.false = s1.next = s.next

i.e s.code = b.code || label b.true || s1.code

Sdd for some control statements:

production	Semantic rules
S->if(b) s1	b.true= newlabel() b.false=s1.next=s.next s.code=b.code||label(b.true` : `) ||s1.code
S->if(b)s1 else s2	b.true=newlabel() b.false=newlabel() S1.next=s2.next=s.next s.code=b.code || label(b.true` : `)|| s1.code ||gen('goto's.next) || label(b.false` : `)|| s2.code

production	Semantic rules
S->while(b) s1	s.begin=newlabel() b.true=newlabel() b.false=s.next s1.next=s.begin s.code=label(s.begin :) || b.code || label (b.true` : `)||s1.code || gen('goto's.begin)

B → E1 rel E2	B.code = E1.code || E2.CODE || GEN('if' E1.addr rel.op E2.addr 'goto' B.true || GEN ('goto' B.false)
B → True	B.code = GEN ('goto' B. True)
B → False	B.code = GEN ('goto' B. False)

Example : if a<b goto B.true
goto B.false

Backpatching:

- while generating code for boolean expressions and flow-of-control statements,a list of jumps are passed as synthesized attributes
- when jump is generated, the target is temporarily not specified
- it is added to a list of jumps without a definite target
- but, all these jumps have the same target
- when the proper label is determined, it is assigned to all the jumps in the list

Switch Statements:

- There is a selector expression which needs to be evaluated followed by a set of values that it can take.
- The expression is evaluated and depending on the value generated particular set of statements are executed
- There is always a set of default statements which is executed if no other value matches the expression

Translation of a switch statement:

```
Code to evaluate E into t
goto test
L1: code for S1
        goto next
L2: code for S2
        goto next
                ...
Ln: code for Sn
        goto next
test:  if t=V1 goto L1
        if t=V2 goto L2    ...
        goto Ln
next:
```

Translation of switch-statement:

- If the number of cases is small say 10 then we use a sequence of conditional jumps
- If the number of values exceeds 10 it is more efficient to construct a hash table for the values with labels of the various statements as entries.

Intermediate code procedures:

In three address code a function call is unraveled into the evaluation of parameters in preparation of a call followed by the call itself.

the statement: n=f(a[i]); is translated to:

1) t1=i * 4
2) t2=a[t1]
3) param t2 /* makes t2 an actual parameter */
4) t3=call f,1
5) n=t3