Module 5

Run-Time Environments:

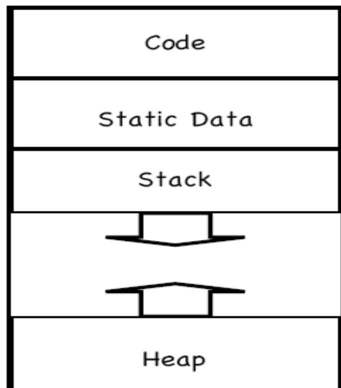
Source Language Issues, Storage Organization, Storage Allocation Strategies

Intermediate Code Generation (ICG):

Intermdiate languages – Graphical representations, Three Address Code, Quadrules, triples, Assignment Statements, Boolean Expressions

Storage Organization

- Run time storage must be divided to hold:
  - Generated target code
  - Data objects – corresponds to a storage location that can hold values.
  - information to keep track of procedure activation



Activation Record

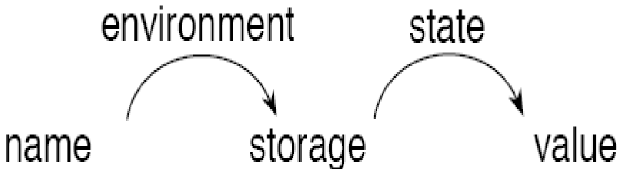
- Information needed for a single execution of a procedure is managed using a contiguous block of storage called activation record or stack frame
- The activation record gets pushed for each procedure call and popped for each procedure return.

return value
actual parameters
[optional] control link
[optional] access link
saved machine status
local data
temporaries

- Temporaries:
  - hold temporary values those arises in the evaluation of expressions.
- Local data:
  - holds data that is local to an execution of a procedure
- Saved m/c status:
  - holds infrmn about the state of m/c just b4 the procedure is called.
  - (values of prgm counter, m/c registers etc that have to be restored when control returns from procedure)
- Acces link (optional):
  - nonlocal data held in other AR
- Control Link (optional):
  - points to the AR of caller
- Actual Parameters:
  - used by the calling procedure to supply parameters to called procedure
- Returned value:
  - used by the called procedure to return value to calling procedure

Name binding

- The ENVIRONMENT is a function mapping from names to storage locations.
- The STATE is a function mapping storage locations to the values held in those locations.



- When an environment maps name x to storage location s, we say "x is BOUND to s". The association is a **NAME BINDING**.
- Assignments change the state, but NOT the environment:
  pi := 3.14

changes the value held in the storage location for pi, but does NOT change the location (the binding) of pi.

Storage Allocation Strategies

- Static Allocation
- Stack Allocation
- Heap Allocation

1. Static Allocation

- Statically allocated names are bound to storage at compile time.
- Since bindings do not change at run time, every time a procedure is activated, its names are bound to the same storage location.
- This ppty allows values of local names to be retained across activations of a procedure.
- That is, when control returns to a procedure, the values of the locals are the same as they were when control left last time.
- The compiler uses the type of a name (retrieved from the symbol table) to determine storage size required.
- The required number of bytes is set aside for the name.
- The address of the storage is fixed at compile time.
- Limitations:

- The size required must be known at compile time.
- Recursive procedures cannot be implemented bcz all activations of a procedure use the same bindings for local names.
- Data structures cannot be created dynamically, since there is no mechanism for storage allocation at run time.

2. Stack Allocation

- Dynamic allocation
- Storage is organized as a stack.
- Activation records are pushed to and popped from stack.
- Storage for the locals in each call of a procedure is contained in the activation record for that call.
- This means locals are bound to fresh storage on every call, bcz a new AR is pushed onto the stack when a call is made.
- The values of the locals are deleted when the activation ends, bcz AR is popped
- ***top*** marks the top of the stack
- To allocate a new activation record, just increment ***top***.
- To deallocate an existing activation record, just decrement ***top*** by the size of record.

Example:

Position in Activation Tree	Activation Records on the Stack	Remarks
s	s a : array	**Frame for a**
s r	s a : array r i : integer	**r is activated**
s r q(1,9)	s a : array q(1,9) i : integer	**Frame for r has been popped and q(1,9) pushed.**
s r q(1,9) p(1,9) q(1,3) p(1,3) q(1,0)	s a : array q(1,9) i : integer q(1,3) i : integer	**Control has just returned to q(1,3)**

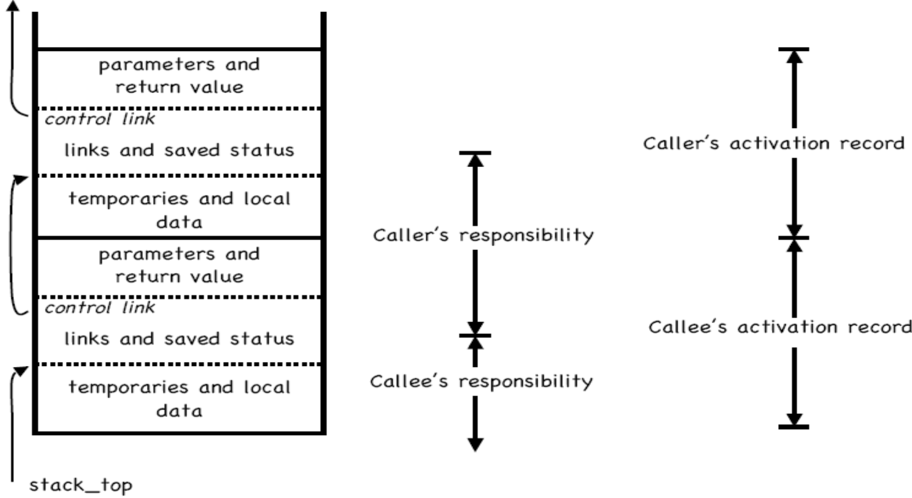
Dashed line in tree represents activations that are ended.

Calling and Return Sequences

- Procedure calls are implemented by generating calling sequences in the target code.

- The **CALLING SEQUENCE** for a procedure allocates an activation record and enters information into its fields.
- The **RETURN SEQUENCE** restores the machine state to allow execution of the calling procedure to continue.
- Some of the calling sequence code is part of the calling procedure(**the caller**), and some is part of the procedure it calls(**the callee**)
- What goes where depends on the language and machine architecture.

Division of task between caller and callee:



Sample calling sequence

1. Caller evaluates the actual parameters and places them into the activation record of the callee.
2. Caller stores a return address and old value for stack_top(top_sp) in the callee's activation record.
3. Caller increments stack_top to the beginning of the temporaries and locals for the callee.
4. Caller branches to the code for the callee.
5. Callee saves all needed register values and status.
6. Callee initializes its locals and begins execution.

Sample return sequence:

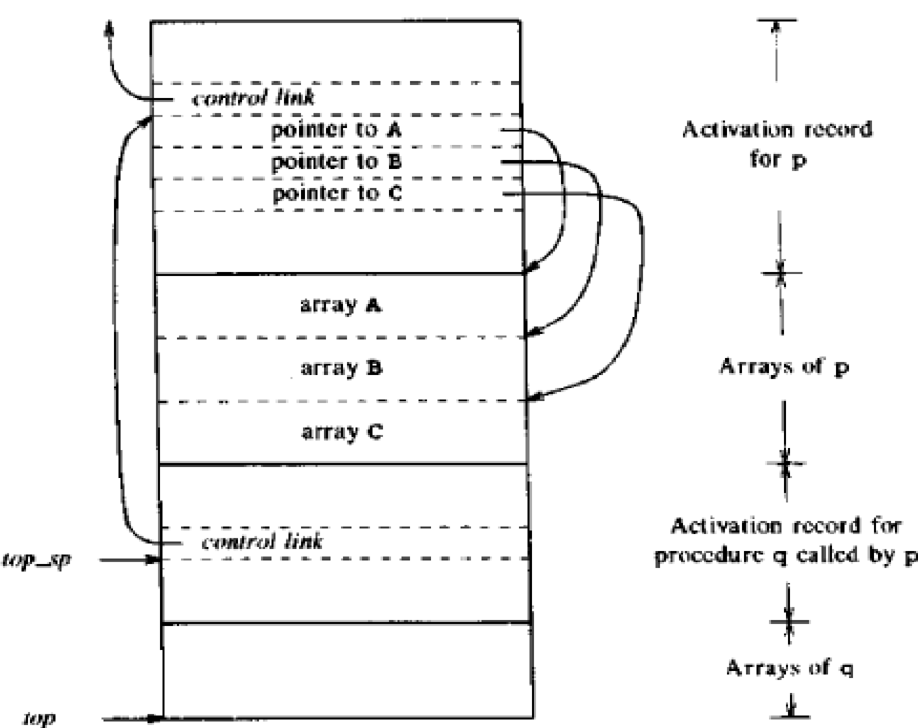
1. Callee places the return value at the correct location in the activation record (next to caller's activation record)
2. Callee uses status information previously saved to restore stack_top and the other registers.

3. Callee branches to the return address previously requested by the caller.
4. Caller copies the return value into its own activation record and uses it to evaluate an expression.

Variable Length Data

- Consider proecedure p has 3 local arrays. The storage for these arrays is not part of AR of p.
- Only a pointer to the beginning of each array appears in AR.
- The relative address of these pointers are known at compile time, so target code can access array elements through pointers.

Access to dynamically allocated arrays:



2 pointers:

1. top

– points to the top of the stack.

2. top_sp

– used to find local data

– Normaly points to the end of machine status field.

Dangling References

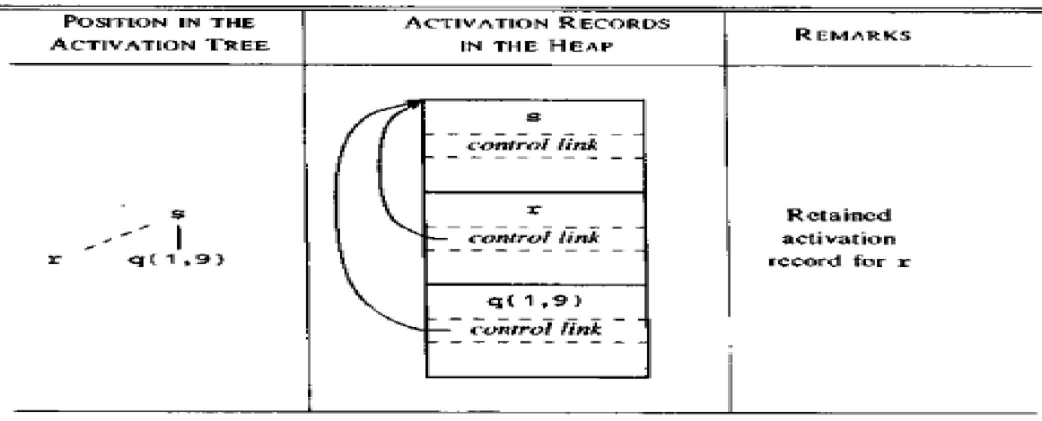
- It occurs when there is a reference to a storage that has been deallocated.
- Example: here use of p is a dangling reference

```
main( )
{
        int *p;
        p = dangle();
}
int *dangle( )
{
        int i = 23;
        return &i;
}
```

- In worst case, if that storage is allocated to some other data, mysterious bugs can appear in prgms with dangling reference.

3. Heap Allocation

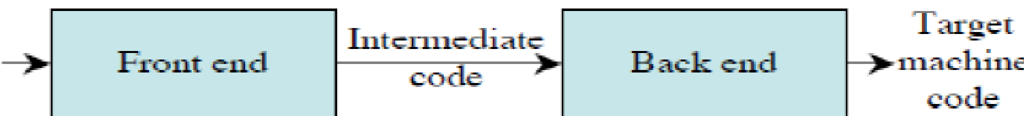
- Stack Allocation cannot be used when:
  - 1. values of local names must be retained when activation ends
  - 2. A called activation outlives caller.(not common)
  - In above cases, deallocation of AR need not be in LIFO order. So storage cannot be organized as stack.
- Heap Allocation:
  - Allocates pieces of contiguous storage, as needed for AR or other objects.
  - Pieces may be deallocated in any order.
  - So over time heap will consist of alternate areas that are free and in use. Its left to the heap manager to make use of this space.



Records for live activations need not be adjacent in a heap.

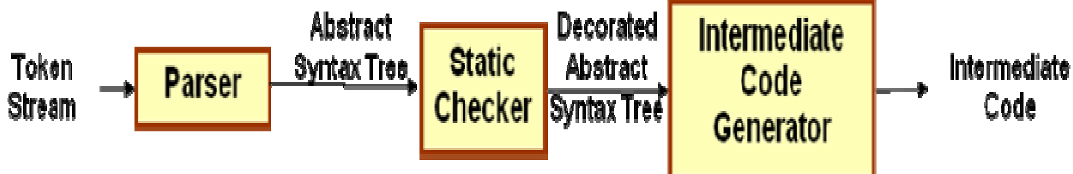
Intermediate Code Generation

- In compiler, the front-end translates a source prgm into an intermediate representation from which the back end generates target code.
- 2 important things:
  - IC Generation process should not be very complex
  - It shouldn't be difficult to produce the target program from the intermediate code.



- A source prgm can be translated directly into the target language, but some benefits of using intermediate form are:
  - 1. Retargetting is facilitated: a compiler for a different machine can be created by attaching a Back-end(which generate Target Code) for the new machine to an existing Front-end (which generate Intermediate Code).
  - 2. A machine Independent Code-Optimizer can be applied to the Intermediate Representation.

Logical Stucture of a Compiler Front End



- **Types of Intermediate Code:**

1. High Level Representation

–Closer to the source language

– Easy to generate from an input program

– Code optimizations may not be straightforward

2. Low Level Representations

– Closer to the target machine

– Suitable for register allocation and instruction selection

–Eeasier for optimizations, final code generation

Intermediate Representations

- Syntax Tree
- DAG (Direct Acyclic Graph)
- Postfix Notation
- 3 Address Code

Syntax Tree or Abstract Syntax Tree(AST)

- Graphical Intermediate Representation
- Syntax Tree depicts the hierarchical structure of a source program.
- Syntax tree (AST) is a condensed form of parse tree useful for representing language constructs.
- In a syntax tree, operators and keywords do not appear as leaves, but appears as intermediate nodes
- In syntax tree, chains of single production may be collapsed.

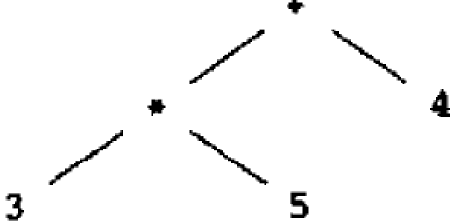


Fig: Syntax Tree

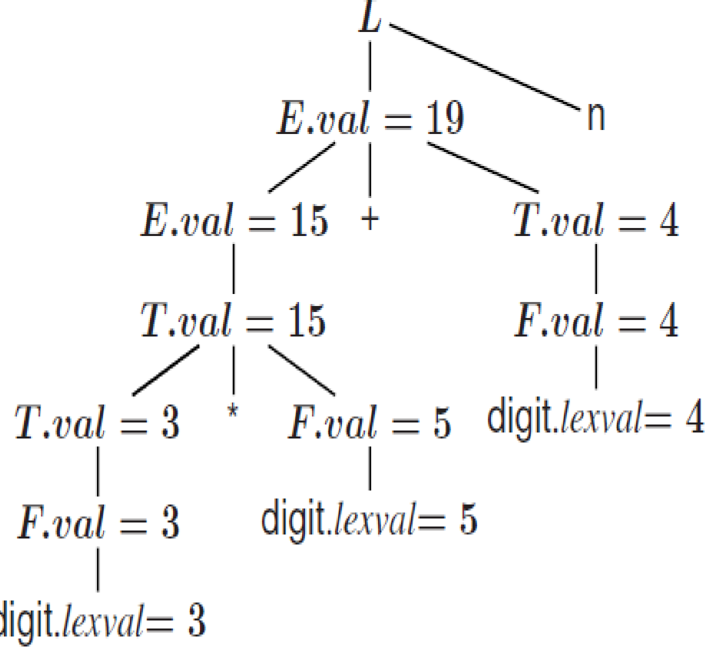
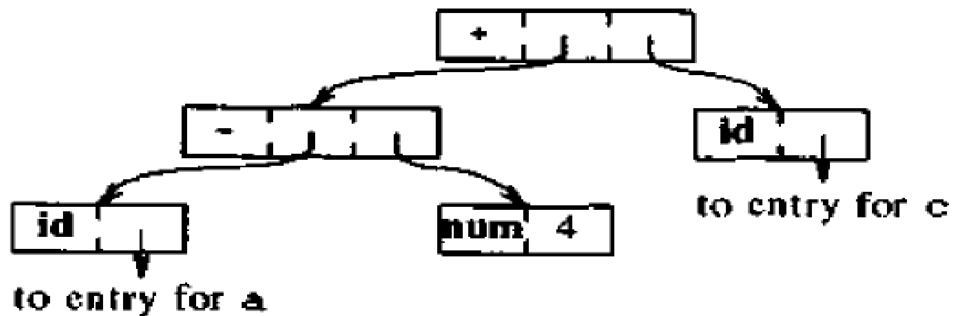


Fig: Annotated Parse Tree for 3 * 5 + 4 n

- Each node in a syntax tree can be implemented as a record with several fields.
- In the node for an operator, one fields identifies the operator and the remaining fields contains the pointers to the nodes for operands



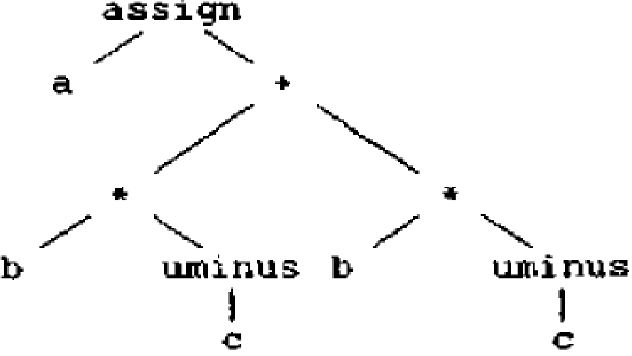
Syntax tree for a-4+c.

- Following functions are used to create the nodes of syntax tree:
  - 1. **mknode(op,left,right):** creates an operator node with label ***op*** and two fields containing pointers to ***left*** and ***right***
  - 2. ***mkleaf(id,entry):*** creates an identifier node with label ***id*** and a field containing ***entry***, a pointer to the symbol table entry for identifier
  - 3. ***mkleaf(num,val):*** creates a number node with label ***num*** and a field containing ***val***, the value of the number.

The following sequence of functions calls creates the syntax tree for the expression a-4+c in Fig. 5.8. In this sequence, $p_1, p_2, \ldots, p_5$ are pointers to nodes, and *entrya* and *entryc* are pointers to the symbol-table entries for identifiers a and c, respectively.

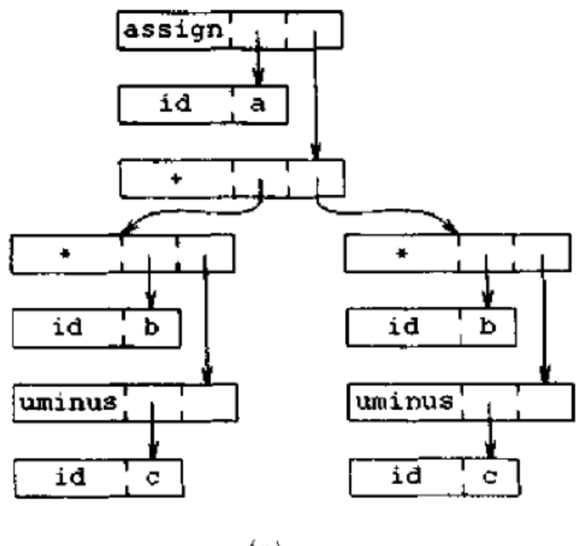
(1) p_1 := *mkleaf* (**id**, *entrya*);
(2) p_2 := *mkleaf* (**num**, 4);
(3) p_3 := *mknode* ('-', p_1, p_2);
(4) p_4 := *mkleaf* (**id**, *entryc*);
(5) p_5 := *mknode* ('+', p_3, p_4);

Ex: a = b * -c + b * -c



(a) Syntax tree.

Two representations of above syntax tree:



(a)

0	id	b	
1	id	c	
2	uminus	1	
3	*	0	2
4	id	b	
5	id	c	
6	uminus	5	
7	*	4	6
8	+	3	7
9	id	a	
10	assign	9	8

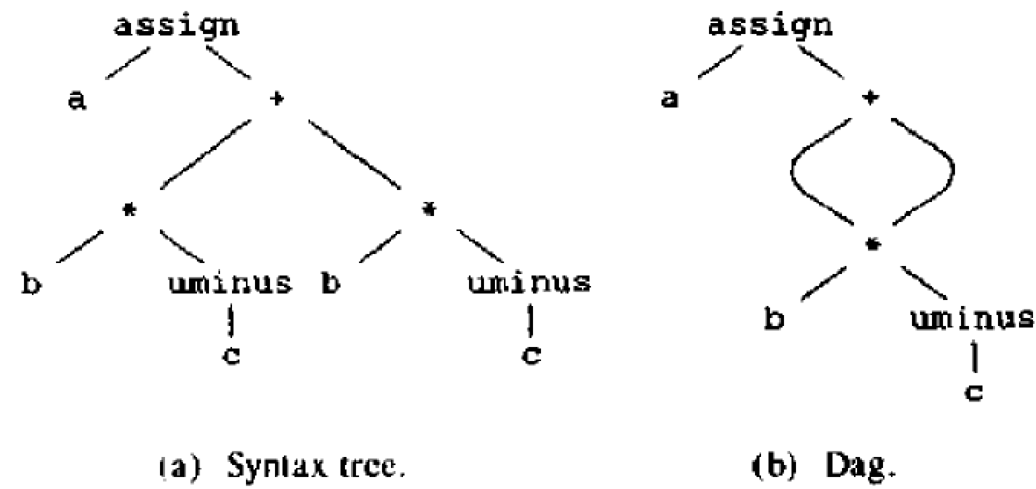
(b)

- Here, nodes are allocated from array of records and the index or position of the node serves as the pointer to the node

Direct Acyclic Graph (DAG)

- Graphical Intermediate Representation
- Dag also gives the hierarchical structure of source pgm but in a more compact way because common sub expressions are identified.

Ex: a=b*-c + b*-c



Graphical representations of a := b * - c + b * - c.

Postfix Notation

- Linearized representation of syntax tree
- In postfix notation, each operator appears immediately after its last operand.
- Operators can be evaluated in the order in which they appear in the string
- Ex:

  Source String : **a := b * -c + b * -c**

  Postfix String: **a b c uminus * b c uminus * + assign**

Postfix Rules:

1. If E is a variable or constant, then the postfix notation for E is E itself

2. If E is an expression of the form ***E_1 op E_2*** then postfix notation for E is ***E_1' E_2' op***, here E_1' and E_2' are the postfix notations for $E_1$and E_2, respectively

3. If E is an expression of the form (E), then the postfix notation for E is the same as the postfix notation for E.

4. For unary operation –E the postfix is E-

Ex: postfix notation for 9- (5+2) is **952+-**

- Postfix notation of an infix expression can be obtained using stack

Three Address Code (TAC)

- 3-Address code is a sequence of statements of the general form:

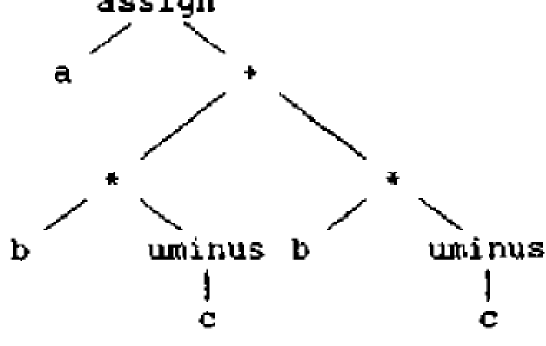
x = y op z , where x,y,z are names, constants or compiler generated temporaries.

- The reason for the term “three address code” is that each statement contains 3 addresses at most. Two for the operands and one for the result.
- Linearized representation of syntax tree
- Ex: x + y * z

3 Address form:

t1 = y * z

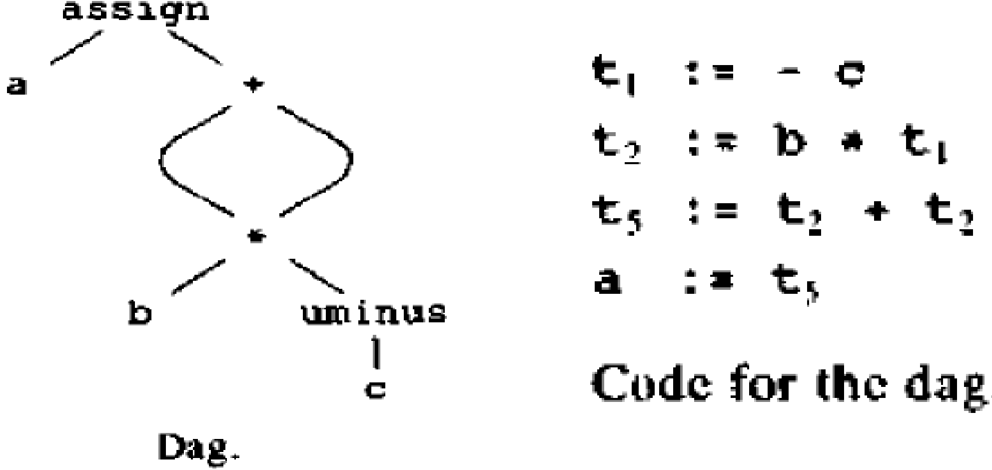
t2 = x + t1



(a) Syntax tree.

```
t1 := - c
t2 := b * t1
t3 := - c
t4 := b * t3
t5 := t2 + t4
a  := t5
```

Code for the syntax tree.



Dag.

Code for the dag.

Types of Three-Address Statements

1. Assignment statements

   – x := y op z, where op is a binary arithmetic or logical operation.

2. Assignment instructions

   – x := op y, where op is a unary operation

3. Copy statements

   – x := y

4.Unconditional jump

   – goto L, TAC with Label L is next to be executed.

5.Conditional jump

   – if x relop y goto L

6. Procedural call and return

   – param x

   – call p,n

   – return y

   – Procedure p(x1,x2,..,xn) is implemented in TAC as:

   param x1

   param x2

   ……….

param xn

call p,n

7.Indexed Assignments

- x = y[i]   or   x[i] = y

8.Address and pointer assignments

- x= &y
- x=*y
- *x =y

SDT into Three Address Code

PRODUCTION	SEMANTIC RULES
$S \rightarrow \textbf{id} := E$	$S.code := E.code \parallel gen(\textbf{id}.place\ ':='\ E.place)$
$E \rightarrow E_1 + E_2$	$E.place := newtemp;$ $E.code := E_1.code \parallel E_2.code \parallel gen(E.place\ ':='\ E_1.place\ '+'\ E_2.place)$
$E \rightarrow E_1 * E_2$	$E.place := newtemp;$ $E.code := E_1.code \parallel E_2.code \parallel gen(E.place\ ':='\ E_1.place\ '*'\ E_2.place)$
$E \rightarrow - E_1$	$E.place := newtemp;$ $E.code := E_1.code \parallel gen(E.place\ ':='\ '\mathtt{uminus}'\ E_1.place)$
$E \rightarrow (E_1)$	$E.place := E_1.place;$ $E.code := E_1.code$
$E \rightarrow \textbf{id}$	$E.place := \textbf{id}.place;$ $E.code := ''$

- The synthesized attribute S.code represents three address code for assignment statement S.
- The nonterminal E has two attributes:
  - 1. E.place : the name that will hold the value of E
  - 2.E.code : the sequence of three-address statements evaluating E
- The function ***newtemp*** returns a new temporary variable on each call

Implementation of Three-Address Statements

- In compiler, three address statements are implemented as records with fields for the operator and operands.
- 3 such representations:

1. Quadruples
2. Triples
3. Indirect Triples

1.Quadruples

- In the quadruple representation, there are four fields for each instruction:
  - *op,arg1, arg2, result*
- Binary ops have the obvious representation
- Unary ops don't use arg2
- Operators like param don't use either arg2 or result

Jumps put the target label into result

The quadruples in Fig (b) implement the three-address code in (a) for the expression

a = b * - c + b * - c

```
t1 = minus c
t2 = b * t1
t3 = minus c
t4 = b * t3
t5 = t2 + t4
 a = t5
```

(a) Three-address code

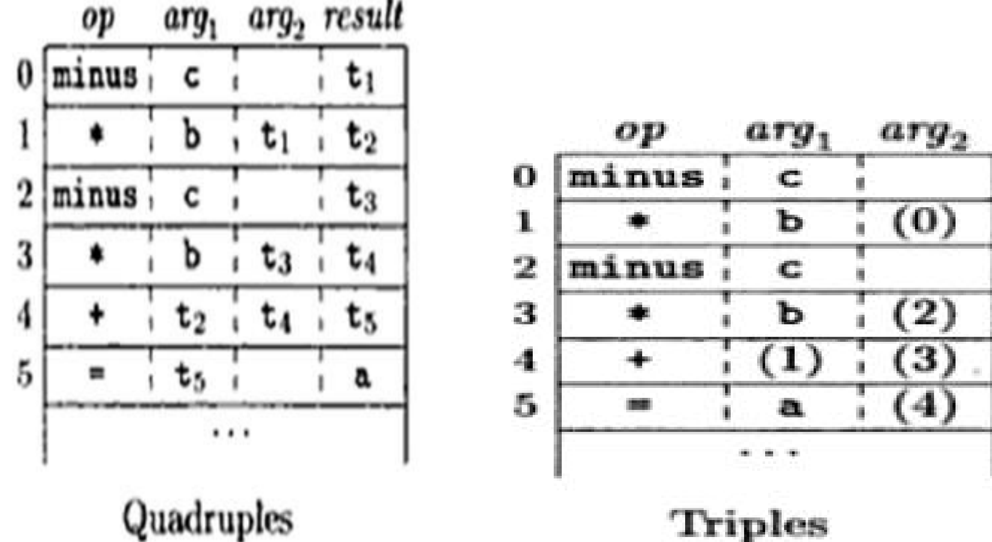
	op	arg_1	arg_2	result
0	minus	c		t_1
1	*	b	t_1	t_2
2	minus	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a
		...		

(b) Quadruples

- The contents of fields arg1, arg2 and result are normally pointers to the symbol table entries for the names represented by these fields.Therefore, temporary names must be entered into symbol table as they created.

2. Triples

- A triple has only three fields for each instruction: *op, arg1, arg2*
- The result of an operation *x op y* is referred to by its position.
- To avoid entering temporary names into the symbol table, we refer a temporary value by the position of statement that computes it.
- Since 3 fields are used, called as triples. Sometimes called as "Two-address code".



	op	arg_1	arg_2	result
0	minus	c		t_1
1	*	b	t_1	t_2
2	minus	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a
	...			

Quadruples

	op	arg_1	arg_2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
	...		

Triples

- Parenthesized numbers represent pointers into the triple structure, while symbol table pointers are represented by the names themselves.

3. **Indirect Triples**

   Consist of a listing pointers to triples, rather than listing the triples themselves.

	statement
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

	op	*arg 1*	*arg 2*
(14)	uminus	c	
(15)	*	b	(14)
(16)	uminus	c	
(17)	*	b	(16)
(18)	+	(15)	(17)
(19)	assign	a	(18)

Indirect triples representation of three-address statements.

Comparison

- When we ultimately produce the target code each temporary and programmer defined name will assign runtime memory location
- This location will be entered into symbol table entry of that data.
- Using the quadruple notation, a three address statement containing a temporary can immediately access the location for that temporary via symbol table.
- But this is not possible with triples notation.
- With quadruple notation, statements can often move around which makes optimization easier.
- This is achieved bcz using quadruple notation the symbol table interposes high degree of indirection between computation of a value and its use.

- With quadruple notation, if we move a statement computing x, the statement using x requires no change.
- But with triples, moving a statement that defines a temporary value requires us to change all references to that statement in arg1 and arg2 arrays. This makes triples difficult to use in optimizing compiler
- With indirect triples also, there is no such problem.
- A statement can be moved by reordering the ***statement list.***

Space Utilization:

- Quadruples and indirect triples requires same amount of space for storage (normal case).
- But if same temporary value is used more than once indirect triples can save some space. This is bcz, 2 or more entries in statement array can point to the same line of o***p-arg1-arg2*** structure.
- Triples requires less space for storage compared to above 2.
- **Quadruples**
    - direct access of the location for temporaries
    - easier for optimization
- **Triples**
    - space efficiency
- **Indirect Triples**
    - easier for optimization
    - space efficiency