MODULE 4 - RUN-TIME ENVIRONMENTS

SOURCE LANGUAGE ISSUES

Procedures:

A *procedure definition* is a declaration that associates an identifier with a statement. The identifier is the *procedure name*, and the statement is the *procedure body*.

For example, the following is the definition of procedure named *readarray* :

```
procedure readarray;
var i : integer;
begin
    for i : = 1 to 9 do read(a[i])
end;
```

When a procedure name appears within an executable statement, the procedure is said to be *called* at that point.

Activation trees:

An *activation tree* is used to depict the way control enters and leaves activations. In an activation tree,

1. Each node represents an activation of a procedure.
2. The root represents the activation of the main program.
3. The node for *a* is the parent of the node for *b* if and only if control flows from activation *a* to *b*.
4. The node for *a* is to the left of the node for *b* if and only if the lifetime of *a* occurs before the lifetime of *b*.

Control stack:

- A *control stack* is used to keep track of live procedure activations. The idea is to push the node for an activation onto the control stack as the activation begins and to pop the node when the activation ends.

- The contents of the control stack are related to paths to the root of the activation tree. When node *n* is at the top of control stack, the stack contains the nodes along the path from *n* to the root.

The Scope of a Declaration:
A declaration is a syntactic construct that associates information with a name.
Declarations may be explicit, such as:

```
var i : integer ;
```

or they may be implicit. Example, any variable name starting with I is assumed to denote an integer.

The portion of the program to which a declaration applies is called the ***scope*** of that declaration.

Binding of names:

Even if each name is declared once in a program, the same name may denote different data objects at run time. "Data object" corresponds to a storage location that holds values.

The term *environment* refers to a function that maps a name to a storage location.
The term *state* refers to a function that maps a storage location to the value held there.

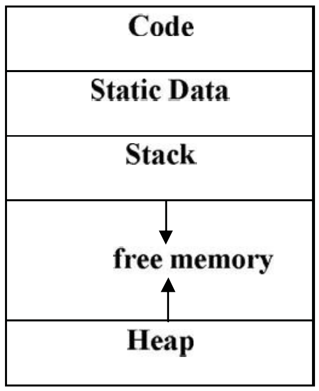
$$\begin{tikzcd}
\text{name} \arrow[r, bend left, "\textit{environment}"] & \text{storage} \arrow[r, bend left, "\textit{state}"] & \text{value}
\end{tikzcd}$$

When an *environment* associates storage location *s* with a name *x*, we say that *x* is *bound* to *s*. This association is referred to as a *binding* of *x*.

STORAGE ORGANISATION

- The executing target program runs in its own logical address space in which each program value has a location.
- The management and organization of this logical address space is shared between the complier, operating system and target machine. The operating system maps the logical address into physical addresses, which are usually spread throughout memory.

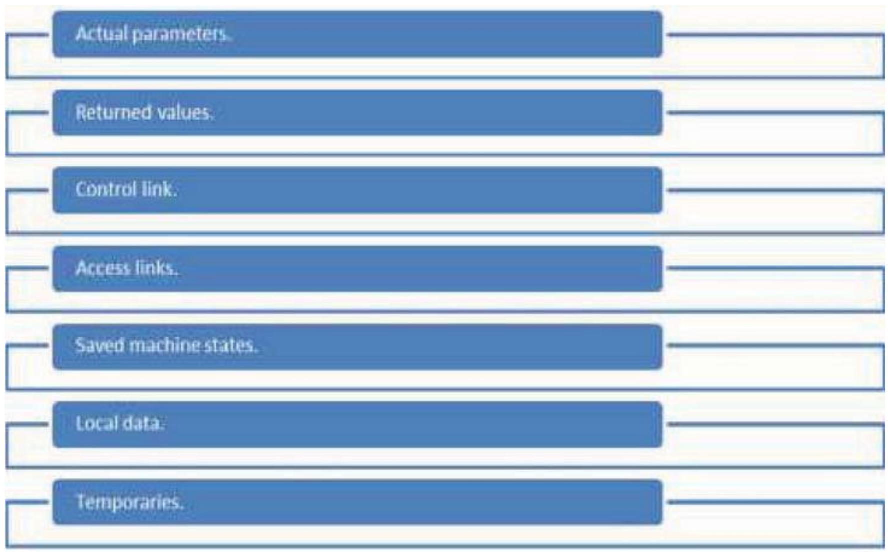
Typical subdivision of run-time memory:



- Run-time storage comes in blocks, where a byte is the smallest unit of addressable memory. Four bytes form a machine word. Multibyte objects are stored in consecutive bytes and given the address of first byte.
- The storage layout for data objects is strongly influenced by the addressing constraints of the target machine.
- A character array of length 10 needs only enough bytes to hold 10 characters, a compiler may allocate 12 bytes to get alignment, leaving 2 bytes unused.
- This unused space due to alignment considerations is referred to as padding.
- The size of some program objects may be known at run time and may be placed in an area called static.
- The dynamic areas used to maximize the utilization of space at run time are stack and heap.

Activation records:

- Procedure calls and returns are usually managed by a run time stack called the *control stack*.
- Each live activation has an activation record on the control stack, with the root of the activation tree at the bottom, the latter activation has its record at the top of the stack.
- The contents of the activation record vary with the language being implemented. The diagram below shows the contents of activation record.



- Temporary values such as those arising from the evaluation of expressions.
- Local data belonging to the procedure whose activation record this is.
- A saved machine status, with information about the state of the machine just before the call to procedures.
- An access link may be needed to locate data needed by the called procedure but found elsewhere.
- A control link pointing to the activation record of the caller.

- Space for the return value of the called functions, if any. Again, not all called procedures return a value, and if one does, we may prefer to place that value in a register for efficiency.
- The actual parameters used by the calling procedure. These are not placed in activation record but rather in registers, when possible, for greater efficiency.

STORAGE ALLOCATION STRATEGIES

The different storage allocation strategies are :

1. **Static allocation –** lays out storage for all data objects at compile time
2. **Stack allocation –** manages the run-time storage as a stack.
3. **Heap allocation –** allocates and deallocates storage as needed at run time from a data area known as heap.

STATIC ALLOCATION

- In static allocation, names are bound to storage as the program is compiled, so there is no need for a run-time support package.
- Since the bindings do not change at run-time, everytime a procedure is activated, its names are bound to the same storage locations.
- Therefore values of local names are *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 the last time.
- From the type of a name, the compiler decides the amount of storage for the name and decides where the activation records go. At compile time, we can fill in the addresses at which the target code can find the data it operates on.

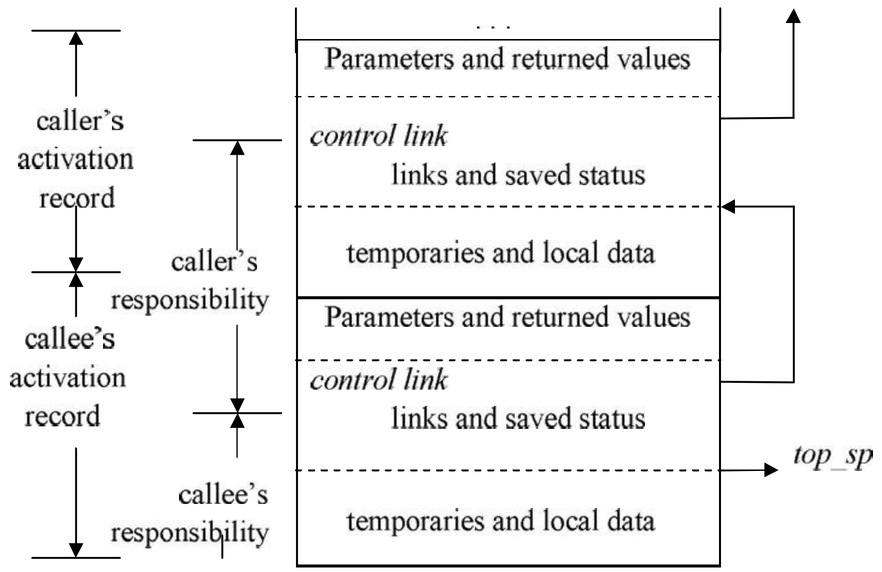
STACK ALLOCATION OF SPACE

- All compilers for languages that use procedures, functions or methods as units of user-defined actions manage at least part of their run-time memory as a stack.
- Each time a procedure is called , space for its local variables is pushed onto a stack, and when the procedure terminates, that space is popped off the stack.

Calling sequences:

- Procedures called are implemented in what is called as calling sequence, which consists of code that allocates an activation record on the stack and enters information into its fields.
- A return sequence is similar to code to restore the state of machine so the calling procedure can continue its execution after the call.
- The code in calling sequence is often divided between the calling procedure (caller) and the procedure it calls (callee).
- When designing calling sequences and the layout of activation records, the following principles are helpful:
  - Values communicated between caller and callee are generally placed at the beginning of the callee's activation record, so they are as close as possible to the caller's activation record.

- Fixed length items are generally placed in the middle. Such items typically include the control link, the access link, and the machine status fields.
- Items whose size may not be known early enough are placed at the end of the activation record. The most common example is dynamically sized array, where the value of one of the callee's parameters determines the length of the array.
- We must locate the top-of-stack pointer judiciously. A common approach is to have it point to the end of fixed-length fields in the activation record. Fixed-length data can then be accessed by fixed offsets, known to the intermediate-code generator, relative to the top-of-stack pointer.


. . .
Parameters and returned values
control link
links and saved status
temporaries and local data
Parameters and returned values
control link
links and saved status
temporaries and local data
caller's activation record
callee's activation record
caller's responsibility
callee's responsibility
top_sp

Division of tasks between caller and callee

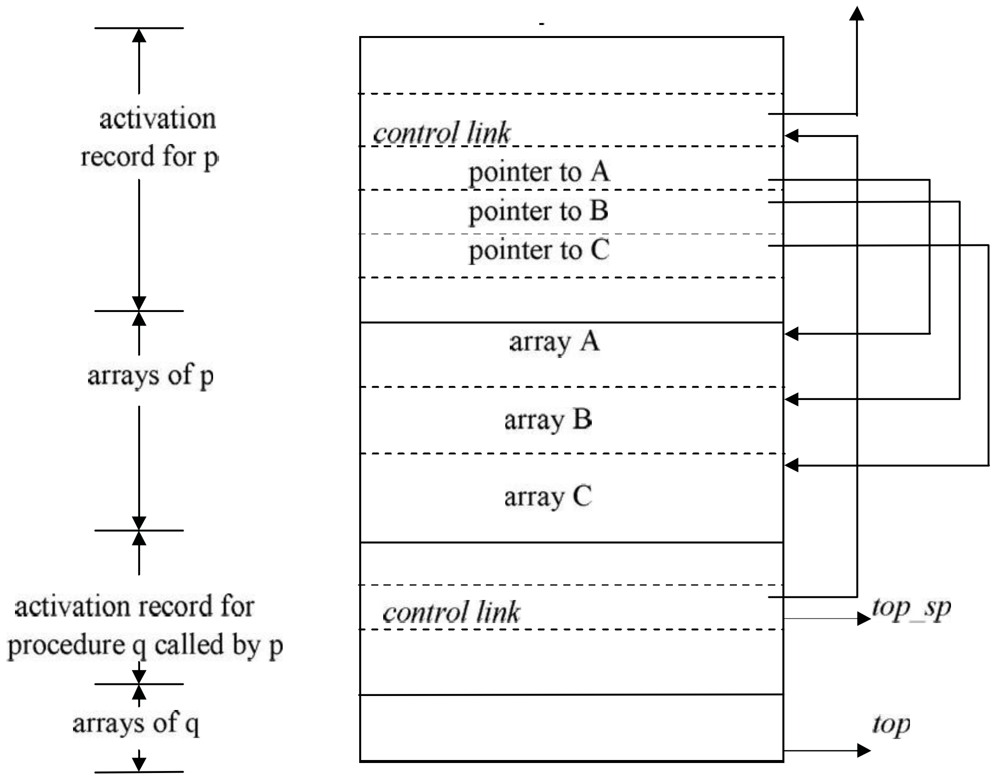
- The calling sequence and its division between caller and callee are as follows.

    - The caller evaluates the actual parameters.
    - The caller stores a return address and the old value of *top_sp* into the callee's activation record. The caller then increments the *top_sp* to the respective positions.
    - The callee saves the register values and other status information.
    - The callee initializes its local data and begins execution.
- A suitable, corresponding return sequence is:

    - The callee places the return value next to the parameters.
    - Using the information in the machine-status field, the callee restores *top_sp* and other registers, and then branches to the return address that the caller placed in the status field.
    - Although *top_sp* has been decremented, the caller knows where the return value is, relative to the current value of *top_sp*; the caller therefore may use that value.

Variable length data on stack:

- The run-time memory management system must deal frequently with the allocation of space for objects, the sizes of which are not known at the compile time, but which are local to a procedure and thus may be allocated on the stack.
- The reason to prefer placing objects on the stack is that we avoid the expense of garbage collecting their space.
- The same scheme works for objects of any type if they are local to the procedure called and have a size that depends on the parameters of the call.



Access to dynamically allocated arrays

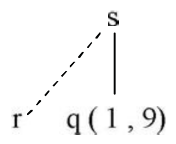
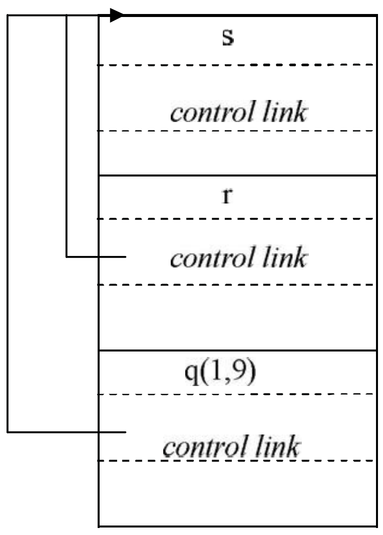
- Procedure p has three local arrays, whose sizes cannot be determined at compile time. The storage for these arrays is not part of the activation record for p.
- Access to the data is through two pointers, *top* and *top-sp*. Here the *top* marks the actual top of stack; it points the position at which the next activation record will begin.
- The second *top-sp* is used to find local, fixed-length fields of the top activation record.
- The code to reposition *top* and *top-sp* can be generated at compile time, in terms of sizes that will become known at run time.

HEAP ALLOCATION

Stack allocation strategy cannot be used if either of the following is possible :

1. The values of local names must be retained when an activation ends.
2. A called activation outlives the caller.

- Heap allocation parcels out pieces of contiguous storage, as needed for activation records or other objects.
- Pieces may be deallocated in any order, so over the time the heap will consist of alternate areas that are free and in use.

Position in the activation tree	Activation records in the heap	Remarks
	 s control link r control link q(1,9) control link	Retained activation record for r

- The record for an activation of procedure r is retained when the activation ends.
- Therefore, the record for the new activation q(1 , 9) cannot follow that for s physically.
- If the retained activation record for r is deallocated, there will be free space in the heap between the activation records for s and q.

MODULE-4 INTERMEDIATE CODE GENERATION

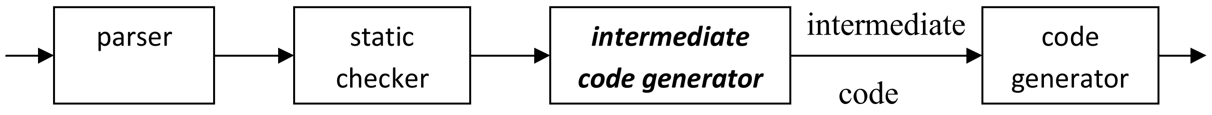
INTRODUCTION

The front end translates a source program into an intermediate representation from which the back end generates target code.

Benefits of using a machine-independent intermediate form are:

1. Retargeting is facilitated. That is, a compiler for a different machine can be created by attaching a back end for the new machine to an existing front end.
2. A machine-independent code optimizer can be applied to the intermediate representation.

Position of intermediate code generator



INTERMEDIATE LANGUAGES

Three ways of intermediate representation:

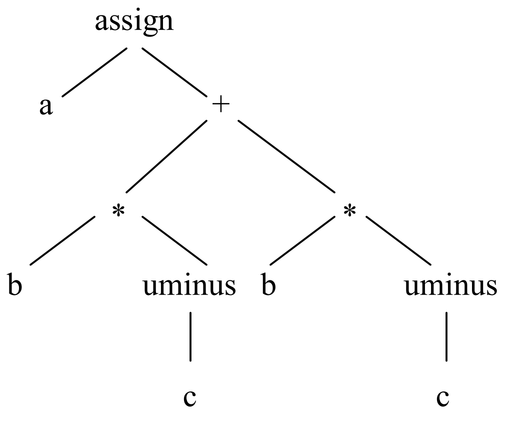
- Syntax tree
- Postfix notation
- Three address code

The semantic rules for generating three-address code from common programming language constructs are similar to those for constructing syntax trees or for generating postfix notation.

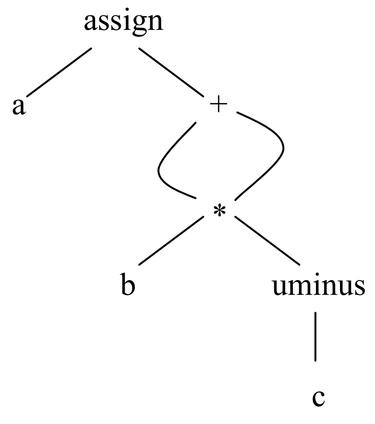
Graphical Representations:

Syntax tree:

A syntax tree depicts the natural hierarchical structure of a source program. A **dag (Directed Acyclic Graph)** gives the same information but in a more compact way because common subexpressions are identified. A syntax tree and dag for the assignment statement **a : = b * - c + b * - c** are as follows:



(a) Syntax tree



(b) Dag

Postfix notation:

Postfix notation is a linearized representation of a syntax tree; it is a list of the nodes of the tree in which a node appears immediately after its children. The postfix notation for the syntax tree given above is

a b c uminus * b c uminus * + assign

Syntax-directed definition:

Syntax trees for assignment statements are produced by the syntax-directed definition. Non-terminal S generates an assignment statement. The two binary operators + and * are examples of the full operator set in a typical language. Operator associativities and precedences are the usual ones, even though they have not been put into the grammar. This definition constructs the tree from the input a : = b * - c + b* - c.

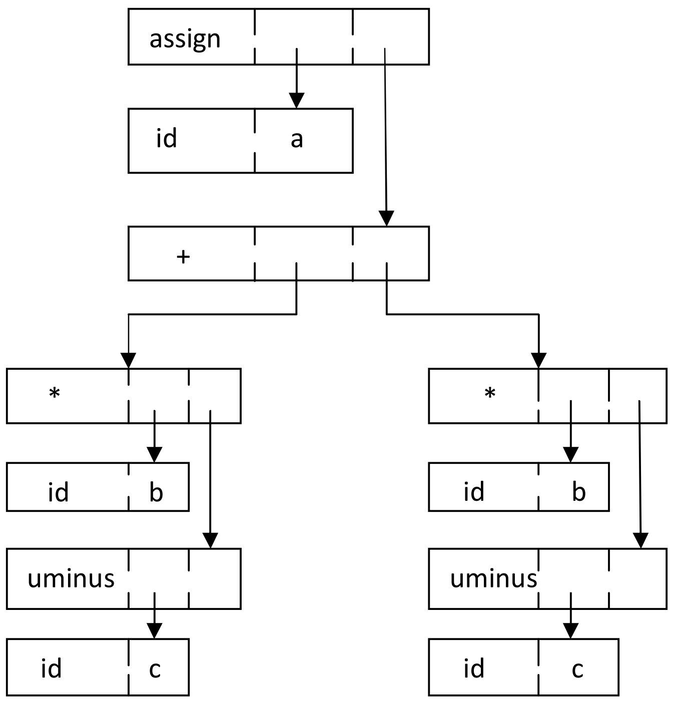
PRODUCTION	SEMANTIC RULE
S → id : = E	S.nptr : = mknode('assign',mkleaf(id, id.place), E.nptr)
E → E_1 + E_2	E.nptr : = mknode('+', E_1.nptr, E_2.nptr)
E → E_1 * E_2	E.nptr : = mknode('*', E_1.nptr, E_2.nptr)
E → - E_1	E.nptr : = mknode('uminus', E_1.nptr)
E → (E_1)	E.nptr : = E_1.nptr
E → id	E.nptr : = mkleaf(id, id.place)

Syntax-directed definition to produce syntax trees for assignment statements

The token **id** has an attribute *place* that points to the symbol-table entry for the identifier. A symbol-table entry can be found from an attribute **id**.*name*, representing the lexeme associated with that occurrence of **id.** If the lexical analyzer holds all lexemes in a single array of characters, then attribute *name* might be the index of the first character of the lexeme.

Two representations of the syntax tree are as follows. In (a) each node is represented as a record with a field for its operator and additional fields for pointers to its children. In (b), nodes are allocated from an array of records and the index or position of the node serves as the pointer to the node. All the nodes in the syntax tree can be visited by following pointers, starting from the root at position 10.

Two representations of the 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)

Three-Address Code:

Three-address code is a sequence of statements of the general form

$$x := y \ op \ z$$

where x, y and z are names, constants, or compiler-generated temporaries; *op* stands for any operator, such as a fixed- or floating-point arithmetic operator, or a logical operator on boolean-valued data. Thus a source language expression like x+ y*z might be translated into a sequence

$$t_1 := y * z$$
$$t_2 := x + t_1$$

where t_1 and t_2 are compiler-generated temporary names.

Advantages of three-address code:

- ➢ The unraveling of complicated arithmetic expressions and of nested flow-of-control statements makes three-address code desirable for target code generation and optimization.
- ➢ The use of names for the intermediate values computed by a program allows three-address code to be easily rearranged – unlike postfix notation.

Three-address code is a linearized representation of a syntax tree or a dag in which explicit names correspond to the interior nodes of the graph. The syntax tree and dag are represented by the three-address code sequences. Variable names can appear directly in three-address statements.

Three-address code corresponding to the syntax tree and dag given above

$t_1 := -c$	$t_1 := -c$
$t_2 := b * t_1$	$t_2 := b * t_1$
$t_3 := -c$	$t_5 := t_2 + t_2$
$t_4 := b * t_3$	$a := t_5$
$t_5 := t_2 + t_4$	
$a := t_5$	
(a) Code for the syntax tree	**(b) Code for the dag**

The reason for the term "three-address code" is that each statement usually contains three addresses, two for the operands and one for the result.

Types of Three-Address Statements:

The common three-address statements are:

1. Assignment statements of the form **x : = y *op* z**, where ***op*** is a binary arithmetic or logical operation.
2. Assignment instructions of the form **x : = *op* y**, where ***op*** is a unary operation. Essential unary operations include unary minus, logical negation, shift operators, and conversion operators that, for example, convert a fixed-point number to a floating-point number.
3. *Copy statements* of the form **x : = y** where the value of *y* is assigned to *x*.
4. The unconditional jump goto L. The three-address statement with label L is the next to be executed.
5. Conditional jumps such as **if *x relop y* goto L**. This instruction applies a relational operator (<, =, >=, etc.) to *x* and *y*, and executes the statement with label L next if *x* stands in relation

relop to y. If not, the three-address statement following if *x relop y* goto L is executed next, as in the usual sequence.

6. *param x* and *call p, n* for procedure calls and *return y*, where y representing a returned value is optional. For example,

```
        param x1
        param x2
          . . .
        param xn
        call p,n
```

generated as part of a call of the procedure $p(x_1, x_2, \ldots, x_n)$.

7. Indexed assignments of the form x : = y[i] and x[i] : = y.

8. Address and pointer assignments of the form x : = &y , x : = *y, and *x : = y.

Syntax-Directed Translation into Three-Address Code:

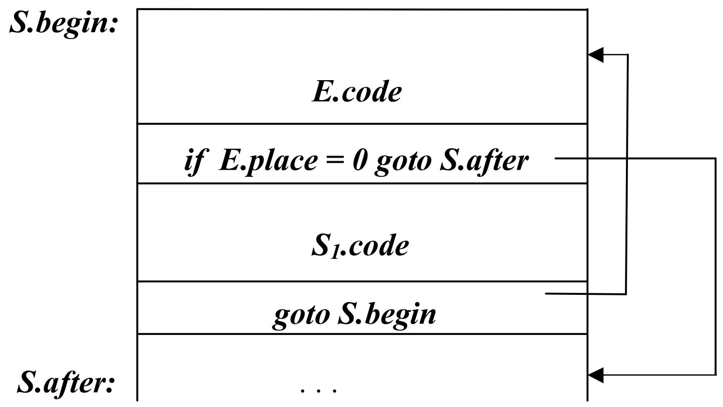
When three-address code is generated, temporary names are made up for the interior nodes of a syntax tree. For example, **id : =** ***E*** consists of code to evaluate ***E*** into some temporary t, followed by the assignment **id**.*place* : = **t.**

Given input a : = b * - c + b * - c, the three-address code is as shown above. The synthesized attribute *S.code* represents the three-address code for the assignment *S.*
The nonterminal *E* has two attributes :
1. *E.place*, the name that will hold the value of *E* , and
2. *E.code*, the sequence of three-address statements evaluating *E*.

Syntax-directed definition to produce three-address code for assignments

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

Semantic rules generating code for a while statement



PRODUCTION	**SEMANTIC RULES**
S → **while** *E* **do** *S₁*	***S.begin := newlabel;*** ***S.after := newlabel;*** ***S.code := gen(S.begin ':')*** || ***E.code*** || ***gen ('if' E.place '=' '0' 'goto' S.after)***|| ***S₁.code*** || ***gen ('goto' S.begin)*** || ***gen (S.after ':')***

- The function *newtemp* returns a sequence of distinct names t_1,t_2,….. in response to successive calls.
- Notation *gen(x ':=' y '+' z)* is used to represent three-address statement x := y + z. Expressions appearing instead of variables like *x, y* and *z* are evaluated when passed to *gen*, and quoted operators or operand, like '+' are taken literally.
- Flow-of–control statements can be added to the language of assignments. The code for ***S*** → **while *E* do *S₁*** is generated using new attributes *S.begin* and *S.after* to mark the first statement in the code for *E* and the statement following the code for S, respectively.
- The function *newlabel* returns a new label every time it is called.
- We assume that a non-zero expression represents true; that is when the value of ***E*** becomes zero, control leaves the while statement.

Implementation of Three-Address Statements:

A three-address statement is an abstract form of intermediate code. In a compiler, these statements can be implemented as records with fields for the operator and the operands. Three such representations are:

- Quadruples
- Triples
- Indirect triples

Quadruples:

- A quadruple is a record structure with four fields, which are, ***op, arg1, arg2*** and ***result.***
- The *op* field contains an internal code for the operator. The three-address statement **x : = y op z** is represented by placing *y* in *arg1*, *z* in *arg2* and *x* in *result.*
- The contents of fields arg1, arg2 and result are normally pointers to the symbol-table entries for the names represented by these fields. If so, temporary names must be entered into the symbol table as they are created.

Triples:

- To avoid entering temporary names into the symbol table, we might refer to a temporary value by the position of the statement that computes it.
- If we do so, three-address statements can be represented by records with only three fields: *op, arg1* and *arg2.*
- The fields *arg1* and *arg2*, for the arguments of *op*, are either pointers to the symbol table or pointers into the triple structure (for temporary values).
- Since three fields are used, this intermediate code format is known as *triples*.

	op	*arg1*	*arg2*	*result*
(0)	uminus	c		t_1
(1)	*	b	t_1	t_2
(2)	uminus	c		t_3
(3)	*	b	t_3	t_4
(4)	+	t_2	t_4	t_5
(5)	: =	t_3		a

(a) Quadruples

	op	*arg1*	*arg2*
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	assign	a	(4)

(b) Triples

Quadruple and triple representation of three-address statements given above

A ternary operation like x[i] : = y requires two entries in the triple structure as shown as below while x : = y[i] is naturally represented as two operations.

	op	*arg1*	*arg2*
(0)	[ ] =	x	i
(1)	assign	(0)	y

(a) x[i] : = y

	op	*arg1*	*arg2*
(0)	= [ ]	y	i
(1)	assign	x	(0)

(b) x : = y[i]

Indirect Triples:

- Another implementation of three-address code is that of listing pointers to triples, rather than listing the triples themselves. This implementation is called indirect triples.
- For example, let us use an array statement to list pointers to triples in the desired order. Then the triples shown above might be represented as follows:

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

	op	*arg1*	*arg2*
(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

DECLARATIONS

As the sequence of declarations in a procedure or block is examined, we can lay out storage for names local to the procedure. For each local name, we create a symbol-table entry with information like the type and the relative address of the storage for the name. The relative address consists of an offset from the base of the static data area or the field for local data in an activation record.

Declarations in a Procedure:

The syntax of languages such as C, Pascal and Fortran, allows all the declarations in a single procedure to be processed as a group. In this case, a global variable, say *offset*, can keep track of the next available relative address.

In the translation scheme shown below:

- Nonterminal P generates a sequence of declarations of the form **id :** ***T.***
- Before the first declaration is considered, *offset* is set to 0. As each new name is seen , that name is entered in the symbol table with offset equal to the current value of *offset*, and *offset* is incremented by the width of the data object denoted by that name.
- The procedure *enter(name, type, offset*) creates a symbol-table entry for *name*, gives its type *type* and relative address *offset* in its data area.
- Attribute *type* represents a type expression constructed from the basic types *integer* and *real* by applying the type constructors *pointer* and *array*. If type expressions are represented by graphs, then attribute *type* might be a pointer to the node representing a type expression.
- The width of an array is obtained by multiplying the width of each element by the number of elements in the array. The width of each pointer is assumed to be 4.

Computing the types and relative addresses of declared names

P → D	***{ offset : = 0 }***
D → D ; D	

D → id : T	***{ enter(id.name, T.type, offset);
	offset : = offset + T.width }***

T → integer	***{ T.type : = integer;
	T.width : = 4 }***

T → real	***{ T.type : = real;
	T.width : = 8 }***

T → array [num] of T_1	***{ T.type : = array(num.val,*** T_1***.type);
	T.width : = num.val X*** T_1***.width }***

T → ↑ T_1	***{ T.type : = pointer (*** T_1***.type);
	T.width : = 4 }***

Keeping Track of Scope Information:

When a nested procedure is seen, processing of declarations in the enclosing procedure is temporarily suspended. This approach will be illustrated by adding semantic rules to the following language:

$P \rightarrow D$

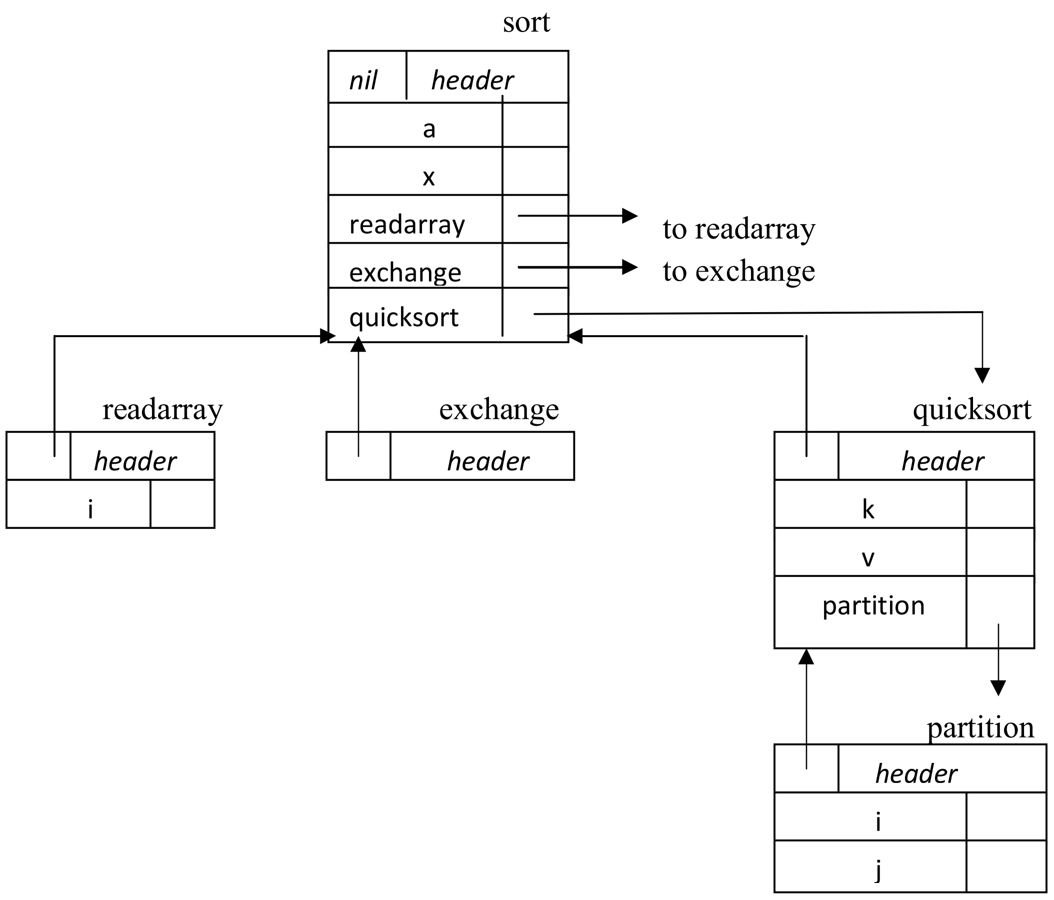
$D \rightarrow D\ ;\ D$ | **id** : T | **proc id** ; D ; S

One possible implementation of a symbol table is a linked list of entries for names.

A new symbol table is created when a procedure declaration $D \rightarrow$ ***proc id*** $D_1;S$ is seen, and entries for the declarations in D_1 are created in the new table. The new table points back to the symbol table of the enclosing procedure; the name represented by id itself is local to the enclosing procedure. The only change from the treatment of variable declarations is that the procedure *enter* is told which symbol table to make an entry in.

For example, consider the symbol tables for procedures *readarray, exchange*, and *quicksort* pointing back to that for the containing procedure *sort*, consisting of the entire program. Since *partition* is declared within *quicksort*, its table points to that of *quicksort*.

Symbol tables for nested procedures



The semantic rules are defined in terms of the following operations:

1. *mktable(previous)* creates a new symbol table and returns a pointer to the new table. The argument *previous* points to a previously created symbol table, presumably that for the enclosing procedure.

2. *enter(table, name, type, offset)* creates a new entry for name *name* in the symbol table pointed to by *table.* Again, *enter* places type *type* and relative address *offset* in fields within the entry.

3. *addwidth(table, width)* records the cumulative width of all the entries in table in the header associated with this symbol table.

4. *enterproc(table, name, newtable)* creates a new entry for procedure *name* in the symbol table pointed to by *table*. The argument *newtable* points to the symbol table for this procedure *name*.

Syntax directed translation scheme for nested procedures

P → M D	***{ addwidth (top(tblptr) , top (offset)); pop (tblptr); pop (offset) }***
M → ε	***{ t : = mktable (nil); push (t,tblptr); push (0,offset) }***
D → D_1 ; D_2	
D → proc id ; N D_1 ; S	***{ t : = top (tblptr); addwidth (t, top (offset)); pop (tblptr); pop (offset); enterproc (top (tblptr), id.name, t) }***
D → id : T	***{ enter (top (tblptr), id.name, T.type, top (offset)); top (offset) := top (offset) + T.width }***
N → ε	***{ t := mktable (top (tblptr)); push (t, tblptr); push (0,offset) }***

- The stack *tblptr* is used to contain pointers to the tables for **sort, quicksort,** and **partition** when the declarations in **partition** are considered.
- The top element of stack *offset* is the next available relative address for a local of the current procedure.
- All semantic actions in the subtrees for B and C in

  A → BC {$action_A$}

  are done before $action_A$ at the end of the production occurs. Hence, the action associated with the marker M is the first to be done.

- The action for nonterminal M initializes stack *tblptr* with a symbol table for the outermost scope, created by operation *mktable(nil).* The action also pushes relative address 0 onto stack offset.

- Similarly, the nonterminal N uses the operation *mktable(top(tblptr))* to create a new symbol table. The argument *top(tblptr)* gives the enclosing scope for the new table.

- For each variable declaration **id:** T, an entry is created for **id** in the current symbol table. The top of stack offset is incremented by T.width.

- When the action on the right side of *D $\rightarrow$ proc id; ND$_1$; S* occurs, the width of all declarations generated by D$_1$ is on the top of stack offset; it is recorded using *addwidth.* Stacks *tblptr* and *offset* are then popped.
  At this point, the name of the enclosed procedure is entered into the symbol table of its enclosing procedure.

ASSIGNMENT STATEMENTS

Suppose that the context in which an assignment appears is given by the following grammar.

P $\rightarrow$ M D

M $\rightarrow$ ε

D $\rightarrow$ D ; D | **id** : T | **proc id** ; N D ; S

N $\rightarrow$ ε

Nonterminal P becomes the new start symbol when these productions are added to those in the translation scheme shown below.

Translation scheme to produce three-address code for assignments

```
S → id : = E        { p : = lookup ( 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 : = newtemp;
                      emit ( E.place ': =' 'uminus' E1.place ) }

E → ( E1 )          { E.place : = E1.place }
```

```
E → id          { p : = lookup ( id.name);
                   if p ≠ nil then
                       E.place : = p
                   else error }
```

Reusing Temporary Names

- The temporaries used to hold intermediate values in expression calculations tend to clutter up the symbol table, and space has to be allocated to hold their values.
- Temporaries can be reused by changing *newtemp*. The code generated by the rules for E → $E_1 + E_2$ has the general form:

  evaluate E_1 into t_1
  evaluate E_2 into t_2
  $t := t_1 + t_2$

- The lifetimes of these temporaries are nested like matching pairs of balanced parentheses.
- Keep a count c , initialized to zero. Whenever a temporary name is used as an operand, decrement c by 1. Whenever a new temporary name is generated, use $c and increase c by 1.
- For example, consider the assignment x := a * b + c * d – e * f

Three-address code with stack temporaries

statement	*value of c*
	0
$0 := a * b	1
$1 := c * d	2
$0 := $0 + $1	1
$1 := e * f	2
$0 := $0 - $1	1
x := $0	0

Addressing Array Elements:

Elements of an array can be accessed quickly if the elements are stored in a block of consecutive locations. If the width of each array element is *w*, then the *i*th element of array A begins in location

$$\textbf{\textit{base}} + (\textbf{\textit{i}} - \textbf{\textit{low}}) \times \textbf{\textit{w}}$$

where *low* is the lower bound on the subscript and *base* is the relative address of the storage allocated for the array. That is, *base* is the relative address of A*[low]*.

The expression can be partially evaluated at compile time if it is rewritten as

$$i \text{ x } w + (\, base - low \text{ x } w)$$

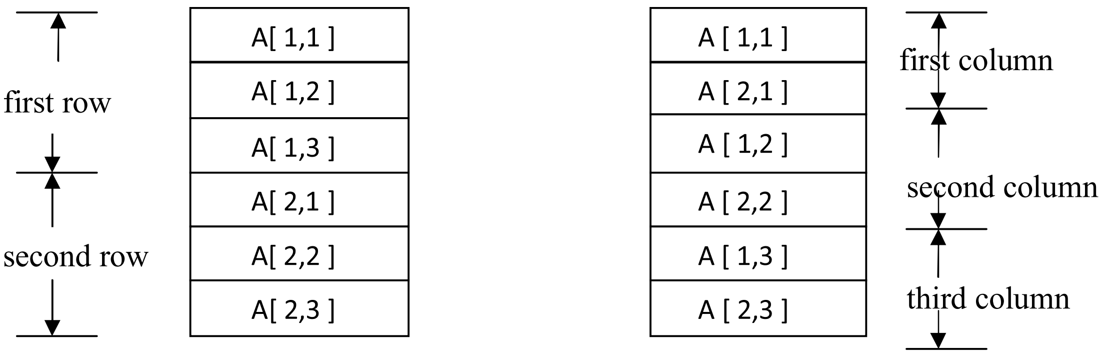
The subexpression $c = base - low$ x w can be evaluated when the declaration of the array is seen. We assume that c is saved in the symbol table entry for A , so the relative address of A[i] is obtained by simply adding i x w to c.

Address calculation of multi-dimensional arrays:

A two-dimensional array is stored in of the two forms :

- Row-major (row-by-row)
- Column-major (column-by-column)

Layouts for a 2 x 3 array



(a) ROW-MAJOR **(b) COLUMN-MAJOR**

In the case of row-major form, the relative address of A[i_1 , i_2] can be calculated by the formula

$$base + ((i_1 - low_1) \text{ x } n_2 + i_2 - low_2) \text{ x } w$$

where, low_1 and low_2 are the lower bounds on the values of i_1 and i_2 and n_2 is the number of values that i_2 can take. That is, if $high_2$ is the upper bound on the value of i_2, then $n_2 = high_2 - low_2 + 1$.

Assuming that i_1 and i_2 are the only values that are known at compile time, we can rewrite the above expression as

$$((\, i_1 \text{ x } n_2\,) + i_2\,) \text{ x } w + (\, base - ((\, low_1 \text{ x } n_2\,) + low_2\,) \text{ x } w)$$

Generalized formula:

The expression generalizes to the following expression for the relative address of A[$i_1,i_2,\ldots,i_k$]

$((\ldots ((\, i_1 n_2 + i_2\,)\, n_3 + i_3) \ldots)\, n_k + i_k\,) \text{ x } w + base - ((\ldots ((low_1 n_2 + low_2) n_3 + low_3) \ldots)$
$n_k + low_k) \text{ x } w$

for all j, $n_j = high_j - low_j + 1$

The Translation Scheme for Addressing Array Elements :

Semantic actions will be added to the grammar :

(1) $S \rightarrow L := E$
(2) $E \rightarrow E + E$
(3) $E \rightarrow (E)$
(4) $E \rightarrow L$
(5) $L \rightarrow Elist\ ]$
(6) $L \rightarrow$ **id**
(7) $Elist \rightarrow Elist , E$
(8) $Elist \rightarrow$ **id** [E

We generate a normal assignment if *L* is a simple name, and an indexed assignment into the location denoted by *L* otherwise :

```
(1)  S → L : = E          {  if L.offset = null then  / * L is a simple id  */
                                emit ( L.place ‘: =’ E.place ) ;
                             else
                                emit ( L.place ‘ [‘ L.offset ‘ ]’  ‘: =’ E.place) }

(2)  E → E1 + E2          { E.place : = newtemp;
                            emit ( E.place ‘: =’ E1.place ‘ +’ E2.place ) }

(3)  E → ( E1 )           { E.place : = E1.place }
```

When an array reference *L* is reduced to *E* , we want the *r*-value of *L*. Therefore we use indexing to obtain the contents of the location *L.place* [*L.offset*] :

```
(4)  E → L                { if L.offset = null  then   /* L is a simple id* /
                                E.place : = L.place
                            else begin
                                E.place : = newtemp;
                                emit ( E.place  ‘: =’ L.place ‘ [‘ L.offset ‘]’)
                            end }

(5)  L → Elist ]          {  L.place : = newtemp;
                             L.offset : = newtemp;
                             emit (L.place ‘: =’ c( Elist.array ));
                             emit (L.offset ‘: =’ Elist.place ‘*’ width (Elist.array)) }

(6)  L → id               {  L.place := id.place;
                              L.offset := null  }

(7)  Elist → Elist1 , 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 }

(8)  Elist → id [ E          { Elist.array : = id.place;

                                  Elist.place : = E.place;
                                  Elist.ndim : = 1 }
```

Type conversion within Assignments :

Consider the grammar for assignment statements as above, but suppose there are two types – real and integer , with integers converted to reals when necessary. We have another attribute *E.type*, whose value is either *real* or *integer*. The semantic rule for *E.type* associated with the production *E* → *E* + *E* is :

```
E → E + E              { E.type : =
                             if E₁.type = integer and
                               E₂.type = integer then integer
                             else real  }
```

The entire semantic rule for *E* → *E* + *E* and most of the other productions must be modified to generate, when necessary, three-address statements of the form x : = inttoreal y, whose effect is to convert integer y to a real of equal value, called x.

Semantic action for $E \rightarrow E_1 + E_2$

```
E.place := newtemp;
if E_1.type = integer and E_2.type = integer then begin
      emit( E.place ': =' E_1.place 'int +' E_2.place);
      E.type : = integer
end
else  if E_1.type = real and E_2.type = real then begin
      emit( E.place ': =' E_1.place 'real +' E_2.place);
      E.type : = real
end
else if E_1.type = integer and E_2.type = real then begin
      u : = newtemp;
      emit( u ': =' 'inttoreal' E_1.place);
      emit( E.place ': =' u ' real +' E_2.place);
      E.type : = real
end
else if E_1.type = real and E_2.type =integer then begin
      u : = newtemp;
      emit( u ': =' 'inttoreal' E_2.place);
      emit( E.place ': =' E_1.place ' real +' u);
      E.type : = real
end
else
      E.type : = type_error;
```

For example, for the input x : = y + i * j
assuming *x* and *y* have type *real*, and i and j have type *integer*, the output would look like

t_1 : = i int* j
t_3 : = inttoreal t_1
t_2 : = y real+ t_3
x : = t_2

BOOLEAN EXPRESSIONS

Boolean expressions have two primary purposes. They are used to compute logical values, but more often they are used as conditional expressions in statements that alter the flow of control, such as if-then-else, or while-do statements.

Boolean expressions are composed of the boolean operators (**and, or,** and **not**) applied to elements that are boolean variables or relational expressions. Relational expressions are of the form E_1 **relop** E_2, where E_1 and E_2 are arithmetic expressions.

Here we consider boolean expressions generated by the following grammar :

E → E **or** E | E **and** E | **not** E | (E) | **id relop id | true | false**

Methods of Translating Boolean Expressions:

There are two principal methods of representing the value of a boolean expression. They are :

- To encode true and false ***numerically*** and to evaluate a boolean expression analogously to an arithmetic expression. Often, 1 is used to denote true and 0 to denote false.
- To implement boolean expressions by ***flow of control***, that is, representing the value of a boolean expression by a position reached in a program. This method is particularly convenient in implementing the boolean expressions in flow-of-control statements, such as the if-then and while-do statements.

Numerical Representation

Here, 1 denotes true and 0 denotes false. Expressions will be evaluated completely from left to right, in a manner similar to arithmetic expressions.

For example :

- The translation for
  a **or** b **and** **not** c
  is the three-address sequence
  t_1 : = **not** c
  t_2 : = b **and** t_1
  t_3 : = a **or** t_2
- A relational expression such as a < b is equivalent to the conditional statement
  if a < b then 1 else 0

which can be translated into the three-address code sequence (again, we arbitrarily start statement numbers at 100) :

```
100 :   if a < b goto 103
101 :   t : = 0
102 :   goto 104
103 :   t : = 1
104 :
```

Translation scheme using a numerical representation for booleans

$E \rightarrow E_1$ **or** E_2	*{ E.place : = newtemp;* *emit(E.place ': =' E_1.place '***or***' E_2.place) }*
$E \rightarrow E_1$ **and** E_2	*{ E.place : = newtemp;* *emit(E.place ': =' E_1.place '***and***' E_2.place) }*
$E \rightarrow$ **not** E_1	*{ E.place : = newtemp;* *emit(E.place ': =' '***not***' E_1.place) }*
$E \rightarrow (E_1)$	*{ E.place : = E_1.place }*
$E \rightarrow$ **id$_1$ relop id$_2$**	*{ E.place : = newtemp;* *emit('if'* **id$_1$***.place* **relop***.op* **id$_2$***.place '***goto***' nextstat +* **3***);* *emit(E.place ': =' '***0***');* *emit('***goto***' nextstat +***2***);* *emit(E.place ': =' '***1***') }*
$E \rightarrow$ **true**	*{ E.place : = newtemp;* *emit(E.place ': =' '***1***') }*
$E \rightarrow$**false**	*{ E.place : = newtemp;* *emit(E.place ': =' '***0***') }*

Short-Circuit Code:

We can also translate a boolean expression into three-address code without generating code for any of the boolean operators and without having the code necessarily evaluate the entire expression. This style of evaluation is sometimes called "**short-circuit**" or "**jumping**" code. It is possible to evaluate boolean expressions without generating code for the boolean operators **and, or,** and **not** if we represent the value of an expression by a position in the code sequence.

Translation of a < b or c < d and e < f

100 : if a < b goto 103	107 : t_2 : = 1
101 : t_1 : = 0	108 : if e < f goto 111
102 : goto 104	109 : t_3 : = 0
103 : t_1 : = 1	110 : goto 112
104 : if c < d goto 107	111 : t_3 : = 1
105 : t_2 : = 0	112 : t_4 : = t_2 and t_3
106 : goto 108	113 : t_5 : = t_1 or t_4

Flow-of-Control Statements

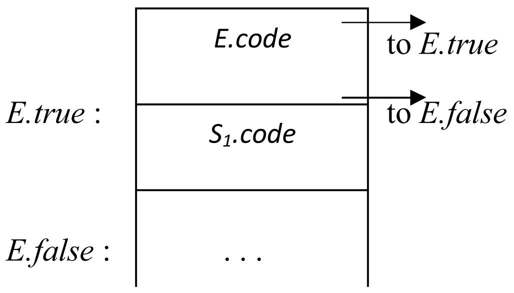
We now consider the translation of boolean expressions into three-address code in the context of if-then, if-then-else, and while-do statements such as those generated by the following grammar:

$S \rightarrow$ **if** E **then** S_1
| **if** E **then** S_1 **else** S_2
| **while** E **do** S_1

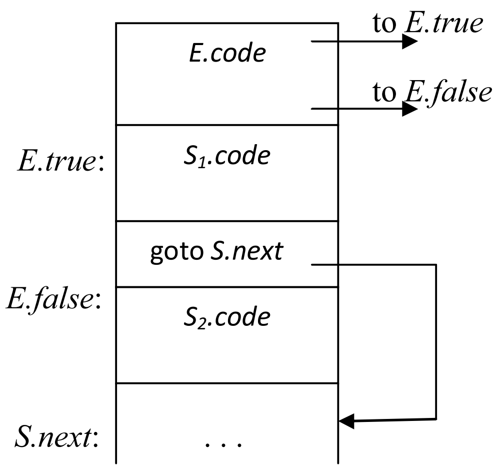
In each of these productions, *E* is the Boolean expression to be translated. In the translation, we assume that a three-address statement can be symbolically labeled, and that the function *newlabel* returns a new symbolic label each time it is called.

- E.true is the label to which control flows if E is true, and E.false is the label to which control flows if E is false.
- The semantic rules for translating a flow-of-control statement S allow control to flow from the translation S.code to the three-address instruction immediately following S.code.
- S.next is a label that is attached to the first three-address instruction to be executed after the code for S.

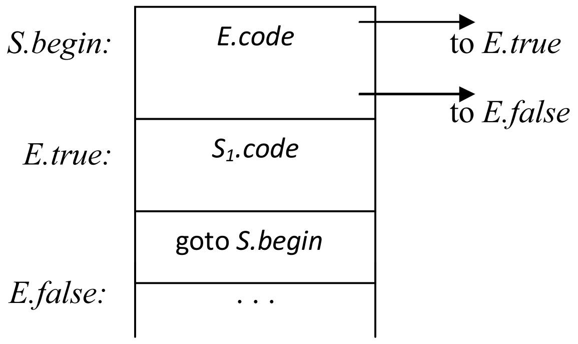
Code for if-then , if-then-else, and while-do statements



(a) if-then



(b) if-then-else



(c) while-do

Syntax-directed definition for flow-of-control statements

PRODUCTION	SEMANTIC RULES
$S \rightarrow$ **if** E **then** S_1	$E.true := newlabel;$ $E.false := S.next;$ $S_1.next := S.next;$ $S.code := E.code \mid\mid gen(E.true\ ':') \mid\mid S_1.code$
$S \rightarrow$ **if** E **then** S_1 **else** S_2	$E.true := newlabel;$ $E.false := newlabel;$ $S_1.next := S.next;$ $S_2.next := S.next;$ $S.code := E.code \mid\mid gen(E.true\ ':') \mid\mid S_1.code \mid\mid$ $gen('\textbf{goto}'\ S.next) \mid\mid$ $gen(E.false\ ':') \mid\mid S_2.code$
$S \rightarrow$ **while** E **do** S_1	$S.begin := newlabel;$ $E.true := newlabel;$ $E.false := S.next;$ $S_1.next := S.begin;$ $S.code := gen(S.begin\ ':') \mid\mid E.code \mid\mid$ $gen(E.true\ ':') \mid\mid S_1.code \mid\mid$ $gen('\textbf{goto}'\ S.begin)$

Control-Flow Translation of Boolean Expressions:

Syntax-directed definition to produce three-address code for booleans

PRODUCTION	SEMANTIC RULES
$E \rightarrow E_1$ **or** E_2	$E_1.true := E.true;$ $E_1.false := newlabel;$ $E_2.true := E.true;$ $E_2.false := E.false;$ $E.code := E_1.code \mid\mid gen(E_1.false\ ':') \mid\mid E_2.code$
$E \rightarrow E_1$ **and** E_2	$E.true := newlabel;$ $E_1.false := E.false;$ $E_2.true := E.true;$ $E_2.false := E.false;$ $E.code := E_1.code \mid\mid gen(E_1.true\ ':') \mid\mid E_2.code$
$E \rightarrow$ **not** E_1	$E_1.true := E.false;$ $E_1.false := E.true;$ $E.code := E_1.code$
$E \rightarrow (E1)$	$E_1.true := E.true;$

	$E_1.false := E.false;$ $E.code := E_1.code$
$E \rightarrow \textbf{id}_1 \textbf{ relop } \textbf{id}_2$	$E.code := gen($‘**if**’ $\textbf{id}_1.place$ **relop.**op $\textbf{id}_2.place$ ‘**goto**’ $E.true)$ || $gen($‘**goto**’ $E.false)$
$E \rightarrow$ **true**	$E.code := gen($‘**goto**’ $E.true)$
$E \rightarrow$ **false**	$E.code := gen($‘**goto**’ $E.false)$

CASE STATEMENTS

The “switch” or “case” statement is available in a variety of languages. The switch-statement syntax is as shown below :

Switch-statement syntax

```
switch expression
        begin
             case value :    statement
             case value :    statement
                      . . .
             case value :    statement
             default :       statement
        end
```

There is a selector expression, which is to be evaluated, followed by *n* constant values that the expression might take, including a default “value” which always matches the expression if no other value does. The intended translation of a switch is code to:

1. Evaluate the expression.
2. Find which value in the list of cases is the same as the value of the expression.
3. Execute the statement associated with the value found.

Step (2) can be implemented in one of several ways :

- By a sequence of conditional **goto** statements, if the number of cases is small.
- By creating a table of pairs, with each pair consisting of a value and a label for the code of the corresponding statement. Compiler generates a loop to compare the value of the expression with each value in the table. If no match is found, the default (last) entry is sure to match.
- If the number of cases s large, it is efficient to construct a hash table.
- There is a common special case in which an efficient implementation of the n-way branch exists. If the values all lie in some small range, say i_{min} to i_{max}, and the number of different values is a reasonable fraction of i_{max} - i_{min}, then we can construct an array of labels, with the label of the statement for value j in the entry of the table with offset j - i_{min} and the label for the default in entries not filled otherwise. To perform switch,

evaluate the expression to obtain the value of j , check the value is within range and transfer to the table entry at offset $j\text{-}i_{min}$.

Syntax-Directed Translation of Case Statements:

Consider the following switch statement:

```
switch E
      begin
            case V1 :     S1
            case V2 :     S2
                  . . .
            case Vn-1 :   Sn-1
            default :     Sn
      end
```

This case statement is translated into intermediate code that has the following form :

Translation of a case statement

```
                 code to evaluate E into t
                 goto test
L1 :             code for S1
                 goto next
L2 :             code for S2
                 goto next
                    . . .
Ln-1 :           code for Sn-1
                 goto next
Ln :             code for Sn
                 goto next
test :            if  t = V1 goto L1
                  if  t = V2 goto L2
                     . . .
                  if  t = Vn-1  goto Ln-1
                  goto Ln
next :
```

To translate into above form :

- When keyword **switch** is seen, two new labels **test** and **next,** and a new temporary **t** are generated.
- As expression E is parsed, the code to evaluate E into **t** is generated. After processing E , the jump **goto test** is generated.
- As each **case** keyword occurs, a new label L_i is created and entered into the symbol table. A pointer to this symbol-table entry and the value V_i of case constant are placed on a stack (used only to store cases).

- Each statement **case** $V_i : S_i$ is processed by emitting the newly created label L_i, followed by the code for S_i , followed by the jump **goto next**.
- Then when the keyword **end** terminating the body of the switch is found, the code can be generated for the n-way branch. Reading the pointer-value pairs on the case stack from the bottom to the top, we can generate a sequence of three-address statements of the form

```
case  V1  L1
case  V2  L2
      . . .
case  Vn-1 Ln-1
case  t  Ln
label next
```

where t is the name holding the value of the selector expression E, and L_n is the label for the default statement.

BACKPATCHING

The easiest way to implement the syntax-directed definitions for boolean expressions is to use two passes. First, construct a syntax tree for the input, and then walk the tree in depth-first order, computing the translations. The main problem with generating code for boolean expressions and flow-of-control statements in a single pass is that during one single pass we may not know the labels that control must go to at the time the jump statements are generated. Hence, a series of branching statements with the targets of the jumps left unspecified is generated. Each statement will be put on a list of goto statements whose labels will be filled in when the proper label can be determined. We call this subsequent filling in of labels ***backpatching***.

To manipulate lists of labels, we use three functions :

1. *makelist*(*i*) creates a new list containing only *i*, an index into the array of quadruples; *makelist* returns a pointer to the list it has made.
2. *merge*(p_1,p_2) concatenates the lists pointed to by p_1 and p_2, and returns a pointer to the concatenated list.
3. *backpatch(p,i)* inserts i as the target label for each of the statements on the list pointed to by *p*.

Boolean Expressions:

We now construct a translation scheme suitable for producing quadruples for boolean expressions during bottom-up parsing. The grammar we use is the following:

(1) $E \rightarrow E_1$ **or** $M\, E_2$
(2) $\quad | \; E_1$ **and** $M\, E_2$
(3) $\quad |$ **not** E_1
(4) $\quad | \; (E_1)$
(5) $\quad |$ $\mathbf{id_1}$ **relop** $\mathbf{id_2}$
(6) $\quad |$ **true**
(7) $\quad |$ **false**
(8) $M \rightarrow \varepsilon$

Synthesized attributes *truelist* and *falselist* of nonterminal E are used to generate jumping code for boolean expressions. Incomplete jumps with unfilled labels are placed on lists pointed to by *E.truelist* and *E.falselist*.

Consider production $E \rightarrow E_1$ **and** $M\,E_2$. If E_1 is false, then E is also false, so the statements on E_1*.falselist* become part of *E.falselist*. If E_1 is true, then we must next test E_2, so the target for the statements E_1*.truelist* must be the beginning of the code generated for E_2. This target is obtained using marker nonterminal M.

Attribute *M.quad* records the number of the first statement of E_2*.code*. With the production $M \rightarrow \varepsilon$ we associate the semantic action

{ *M.quad* : = *nextquad* }

The variable *nextquad* holds the index of the next quadruple to follow. This value will be backpatched onto the E_1*.truelist* when we have seen the remainder of the production $E \rightarrow E_1$ **and** $M\,E_2$. The translation scheme is as follows:

(1) $E \rightarrow E_1$ **or** $M\,E_2$	{ *backpatch* (E_1*.falselist*, *M.quad*);
	E.truelist : = *merge*(E_1*.truelist*, E_2*.truelist*);
	E.falselist : = E_2*.falselist* }

(2) $E \rightarrow E_1$ **and** $M\,E_2$	{ *backpatch* (E_1*.truelist*, *M.quad*);
	E.truelist : = E_2*.truelist*;
	E.falselist : = *merge*(E_1*.falselist*, E_2*.falselist*) }

(3) $E \rightarrow$ **not** E_1	{ *E.truelist* : = E_1*.falselist*; *E.falselist* : = E_1*.truelist*; }
(4) $E \rightarrow (E_1)$	{ *E.truelist* : = E_1*.truelist*; *E.falselist* : = E_1*.falselist*; }

(5) $E \rightarrow$ **id$_1$ relop id$_2$**	{ *E.truelist* : = *makelist* (*nextquad*);
	E.falselist : = *makelist*(*nextquad* + *1*);
	emit('if' **id$_1$**.*place* **relop**.*op* **id$_2$**.place '**goto**_')
	emit('**goto**_') }

(6) $E \rightarrow$ **true**	{ *E.truelist* : = *makelist*(*nextquad*);
	emit('**goto**_') }

(7) $E \rightarrow$ **false**	{ *E.falselist* : = *makelist*(*nextquad*);
	emit('**goto**_') }

(8) $M \rightarrow \varepsilon$	{ *M.quad* : = *nextquad* }

Flow-of-Control Statements:

A translation scheme is developed for statements generated by the following grammar :

(1) $S \rightarrow$ **if** E **then** S
(2) $\mid$ **if** E **then** S **else** S
(3) $\mid$ **while** E **do** S
(4) $\mid$ **begin** L **end**
(5) $\mid$ A
(6) $L \rightarrow L \,;\, S$
(7) $\mid$ S

Here S denotes a statement, L a statement list, A an assignment statement, and E a boolean expression. We make the tacit assumption that the code that follows a given statement in execution also follows it physically in the quadruple array. Else, an explicit jump must be provided.

Scheme to implement the Translation:

The nonterminal E has two attributes *E.truelist* and *E.falselist*. *L* and *S* also need a list of unfilled quadruples that must eventually be completed by backpatching. These lists are pointed to by the attributes *L..nextlist* and *S.nextlist*. *S.nextlist* is a pointer to a list of all conditional and unconditional jumps to the quadruple following the statement S in execution order, and *L.nextlist* is defined similarly.

The semantic rules for the revised grammar are as follows:

(1) $S \rightarrow$ **if** E **then** $M_1\, S_1\, N$ **else** $M_2\, S_2$
{ *backpatch* (*E.truelist*, M_1*.quad*);
backpatch (*E.falselist*, M_2*.quad*);
S.nextlist : = *merge* (S_1*.nextlist*, *merge* (*N.nextlist*, S_2*.nextlist*)) }

We backpatch the jumps when E is true to the quadruple M_1*.quad*, which is the beginning of the code for S_1. Similarly, we backpatch jumps when E is false to go to the beginning of the code for S_2. The list *S.nextlist* includes all jumps out of S_1 and S_2, as well as the jump generated by *N*.

(2) $N \rightarrow \varepsilon$ { *N.nextlist* : = *makelist*(*nextquad*);
emit('**goto** _') }

(3) $M \rightarrow \varepsilon$ { *M.quad* : = *nextquad* }

(4) $S \rightarrow$ **if** E **then** $M\, S_1$ { *backpatch*(*E.truelist*, *M.quad*);
S.nextlist : = *merge*(*E.falselist*, S_1*.nextlist*) }

(5) $S \rightarrow$ **while** $M_1\, E$ **do** $M_2\, S_1$ { *backpatch*(S_1*.nextlist*, M_1*.quad*);
backpatch(*E.truelist*, M_2*.quad*);
S.nextlist : = *E.falselist*
emit('**goto**' M_1*.quad*) }

(6) $S \rightarrow$ **begin** L **end** { *S.nextlist* : = *L.nextlist* }

(7) $S \rightarrow A$ { *S.nextlist* : = **nil** }

The assignment *S.nextlist* : = **nil** initializes *S.nextlist* to an empty list.

(8) $L \rightarrow L1 \; ; M \, S$ { *backpatch*(L_1*.nextlist*, *M.quad*);
L.nextlist : = *S.nextlist* }

The statement following L_1 in order of execution is the beginning of *S*. Thus the *L1.nextlist* list is backpatched to the beginning of the code for *S*, which is given by *M.quad.*

(9) $L \rightarrow S$ { *L.nextlist* : = *S.nextlist* }

PROCEDURE CALLS

The procedure is such an important and frequently used programming construct that it is imperative for a compiler to generate good code for procedure calls and returns. The run-time routines that handle procedure argument passing, calls and returns are part of the run-time support package.

Let us consider a grammar for a simple procedure call statement

(1) $S \rightarrow$ **call id** (*Elist*)
(2) $Elist \rightarrow Elist \, , E$
(3) $Elist \rightarrow E$

Calling Sequences:

The translation for a call includes a calling sequence, a sequence of actions taken on entry to and exit from each procedure. The falling are the actions that take place in a calling sequence :

- When a procedure call occurs, space must be allocated for the activation record of the called procedure.
- The arguments of the called procedure must be evaluated and made available to the called procedure in a known place.
- Environment pointers must be established to enable the called procedure to access data in enclosing blocks.
- The state of the calling procedure must be saved so it can resume execution after the call.
- Also saved in a known place is the return address, the location to which the called routine must transfer after it is finished.
- Finally a jump to the beginning of the code for the called procedure must be generated.

For example, consider the following syntax-directed translation

(1) $S \rightarrow$ **call id** (*Elist*)
{ **for** each item *p* on *queue* **do**
emit (' **param** ' *p*);

emit (‘call’ **id.***place*) }

(2) *Elist* → *Elist* , *E*

{ append *E.place* to the end of *queue* }

(3) *Elist* → *E*

{ initialize *queue* to contain only *E.place* }

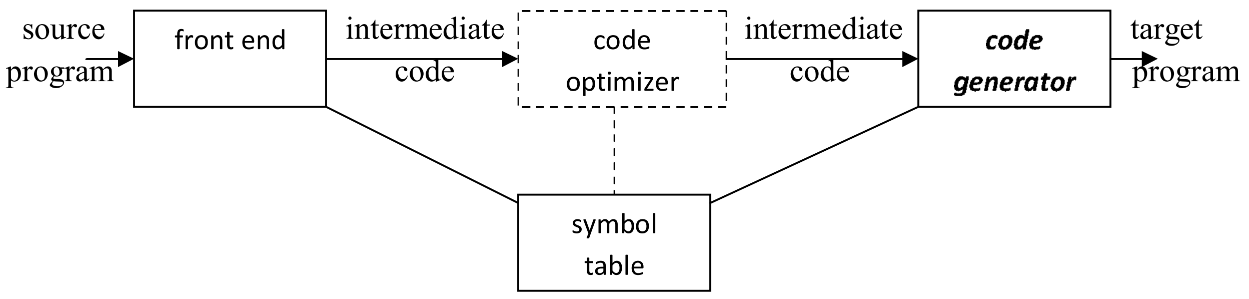
- Here, the code for S is the code for *Elist*, which evaluates the arguments, followed by a **param** *p* statement for each argument, followed by a **call** statement.

- *queue* is emptied and then gets a single pointer to the symbol table location for the name that denotes the value of E.

MODULE-4 CODE GENERATION

The final phase in compiler model is the code generator. It takes as input an intermediate representation of the source program and produces as output an equivalent target program. The code generation techniques presented below can be used whether or not an optimizing phase occurs before code generation.

Position of code generator



ISSUES IN THE DESIGN OF A CODE GENERATOR

The following issues arise during the code generation phase :

1. Input to code generator
2. Target program
3. Memory management
4. Instruction selection
5. Register allocation
6. Evaluation order

1. Input to code generator:

- The input to the code generation consists of the intermediate representation of the source program produced by front end , together with information in the symbol table to determine run-time addresses of the data objects denoted by the names in the intermediate representation.

- Intermediate representation can be :
  a. Linear representation such as postfix notation
  b. Three address representation such as quadruples
  c. Virtual machine representation such as stack machine code
  d. Graphical representations such as syntax trees and dags.

- Prior to code generation, the front end must be scanned, parsed and translated into intermediate representation along with necessary type checking. Therefore, input to code generation is assumed to be error-free.

2. Target program:

- The output of the code generator is the target program. The output may be :
  a. Absolute machine language
     - It can be placed in a fixed memory location and can be executed immediately.

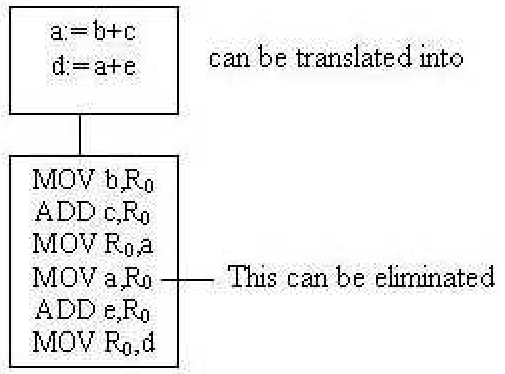
b. Relocatable machine language
   - It allows subprograms to be compiled separately.

c. Assembly language
   - Code generation is made easier.

3. **Memory management:**
   - Names in the source program are mapped to addresses of data objects in run-time memory by the front end and code generator.
   - It makes use of symbol table, that is, a name in a three-address statement refers to a symbol-table entry for the name.
   - Labels in three-address statements have to be converted to addresses of instructions. For example,

     j : **goto** *i* generates jump instruction as follows :
     - if $i < j$, a backward jump instruction with target address equal to location of code for quadruple *i* is generated.
     - if $i > j$, the jump is forward. We must store on a list for quadruple *i* the location of the first machine instruction generated for quadruple *j*. When *i* is processed, the machine locations for all instructions that forward jumps to *i* are filled.

4. **Instruction selection:**
   - The instructions of target machine should be complete and uniform.
   - Instruction speeds and machine idioms are important factors when efficiency of target program is considered.
   - The quality of the generated code is determined by its speed and size.
   - The former statement can be translated into the latter statement as shown below:



5. **Register allocation**
   - Instructions involving register operands are shorter and faster than those involving operands in memory.
   - The use of registers is subdivided into two subproblems :
     - ***Register allocation*** – the set of variables that will reside in registers at a point in the program is selected.

- ➢ ***Register assignment*** – the specific register that a variable will reside in is picked.

- Certain machine requires even-odd *register pairs* for some operands and results. For example , consider the division instruction of the form :

  D x, y

  where, x – dividend even register in even/odd register pair
  y – divisor
  even register holds the remainder
  odd register holds the quotient

6. Evaluation order

- The order in which the computations are performed can affect the efficiency of the target code. Some computation orders require fewer registers to hold intermediate results than others.

TARGET MACHINE

- Familiarity with the target machine and its instruction set is a prerequisite for designing a good code generator.
- The target computer is a byte-addressable machine with 4 bytes to a word.
- It has *n* general-purpose registers, $R_0, R_1, \ldots, R_{n-1}$.
- It has two-address instructions of the form:

  op source, destination

  where, *op* is an op-code, and *source* and *destination* are data fields.

- It has the following op-codes :

  MOV (move *source* to *destination*)
  ADD (add *source* to *destination*)
  SUB (subtract *source* from *destination*)

- The *source* and *destination* of an instruction are specified by combining registers and memory locations with address modes.

Address modes with their assembly-language forms

MODE	FORM	*ADDRESS*	ADDED COST
absolute	M	M	1
register	R	R	0
indexed	*c*(R)	*c*+*contents*(R)	1
indirect register	*R	*contents* (R)	0
indirect indexed	**c*(R)	*contents*(*c*+ *contents*(R))	1
literal	#*c*	*c*	1

- For example : MOV R_0, M stores contents of Register R_0 into memory location M ;
  MOV 4(R_0), M stores the value *contents*(4+*contents*(R_0)) into M.

Instruction costs :

- Instruction cost = 1+cost for source and destination address modes. This cost corresponds to the length of the instruction.
- Address modes involving registers have cost zero.
- Address modes involving memory location or literal have cost one.
- Instruction length should be minimized if space is important. Doing so also minimizes the time taken to fetch and perform the instruction.
  For example : MOV R0, R1 copies the contents of register R0 into R1. It has cost one, since it occupies only one word of memory.
- The three-address statement **a : = b + c** can be implemented by many different instruction sequences :

```
i) MOV b, R0
   ADD c, R0              cost = 6
   MOV R0, a

ii) MOV b, a
    ADD c, a              cost = 6

iii) Assuming R0, R1 and R2 contain the addresses of a, b, and c :
     MOV *R1, *R0
     ADD *R2, *R0         cost = 2
```

- In order to generate good code for target machine, we must utilize its addressing capabilities efficiently.

RUN-TIME STORAGE MANAGEMENT

- Information needed during an execution of a procedure is kept in a block of storage called an activation record, which includes storage for names local to the procedure.
- The two standard storage allocation strategies are:
  1. Static allocation
  2. Stack allocation
- In static allocation, the position of an activation record in memory is fixed at compile time.
- In stack allocation, a new activation record is pushed onto the stack for each execution of a procedure. The record is popped when the activation ends.
- The following three-address statements are associated with the run-time allocation and deallocation of activation records:
  1. Call,
  2. Return,
  3. Halt, and
  4. Action, a placeholder for other statements.
- We assume that the run-time memory is divided into areas for:
  1. Code
  2. Static data
  3. Stack

Static allocation

Implementation of call statement:

The codes needed to implement static allocation are as follows:

MOV *#here* + 20, *callee.static_area* /*It saves return address*/

GOTO *callee.code_area* /*It transfers control to the target code for the called procedure */

where,
callee.static_area – Address of the activation record
callee.code_area – Address of the first instruction for called procedure
#here + 20 – Literal return address which is the address of the instruction following GOTO.

Implementation of return statement:

A return from procedure *callee* is implemented by :

GOTO **callee.static_area*

This transfers control to the address saved at the beginning of the activation record.

Implementation of action statement:

The instruction ACTION is used to implement action statement.

Implementation of halt statement:

The statement HALT is the final instruction that returns control to the operating system.

Stack allocation

Static allocation can become stack allocation by using relative addresses for storage in activation records. In stack allocation, the position of activation record is stored in register so words in activation records can be accessed as offsets from the value in this register.

The codes needed to implement stack allocation are as follows:

Initialization of stack:

MOV *#stackstart* , SP /* initializes stack */

Code for the first procedure

HALT /* terminate execution */

Implementation of Call statement:

ADD *#caller.recordsize*, SP /* increment stack pointer */

MOV *#here* + 16, *SP /*Save return address */

GOTO *callee.code_area*

where,
caller.recordsize – size of the activation record
#here + 16 – address of the instruction following the **GOTO**

Implementation of Return statement:

GOTO *0 (SP) /*return to the caller */

SUB *#caller.recordsize*, SP /* decrement SP and restore to previous value */

BASIC BLOCKS AND FLOW GRAPHS

Basic Blocks

- A *basic block* is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without any halt or possibility of branching except at the end.
- The following sequence of three-address statements forms a basic block:
  t_1 : = a * a
  t_2 : = a * b
  t_3 : = 2 * t_2
  t_4 : = t_1 + t_3
  t_5 : = b * b
  t_6 : = t_4 + t_5

Basic Block Construction:

Algorithm: Partition into basic blocks

Input: A sequence of three-address statements

Output: A list of basic blocks with each three-address statement in exactly one block

Method:

1. We first determine the set of *leaders*, the first statements of basic blocks. The rules we use are of the following:
   a. The first statement is a leader.
   b. Any statement that is the target of a conditional or unconditional goto is a leader.
   c. Any statement that immediately follows a goto or conditional goto statement is a leader.
2. For each leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of the program.

- Consider the following source code for dot product of two vectors a and b of length 20

```
begin
          prod :=0;
          i:=1;
          do begin
                    prod :=prod+ a[i] * b[i];
                    i :=i+1;
          end
          while i <= 20
end
```

- The three-address code for the above source program is given as :

```
(1)     prod := 0
(2)     i := 1
(3)     t1 := 4* i
(4)     t2 := a[t1]        /*compute a[i] */
(5)     t3 := 4* i
(6)     t4 :=  b[t3]       /*compute b[i] */
(7)     t5 := t2*t4
(8)     t6 := prod+t5
(9)     prod := t6
(10)    t7 := i+1
(11)    i := t7
(12)    if i<=20 goto (3)
```

Basic block 1: Statement (1) to (2)

Basic block 2: Statement (3) to (12)

Transformations on Basic Blocks:

A number of transformations can be applied to a basic block without changing the set of expressions computed by the block. Two important classes of transformation are :

- Structure-preserving transformations
- Algebraic transformations

1. Structure preserving transformations:

a) Common subexpression elimination:

$$\begin{array}{lcl} a := b + c & & a := b + c \\ b := a - d & \longrightarrow & b := a - d \\ c := b + c & & c := b + c \\ d := a - d & & d := b \end{array}$$

Since the second and fourth expressions compute the same expression, the basic block can be transformed as above.

b) Dead-code elimination:

Suppose *x* is dead, that is, never subsequently used, at the point where the statement x : = y + z appears in a basic block. Then this statement may be safely removed without changing the value of the basic block.

c) Renaming temporary variables:

A statement **t : = b + c** (t is a temporary) can be changed to **u : = b + c** (u is a new temporary) and all uses of this instance of **t** can be changed to **u** without changing the value of the basic block.
Such a block is called a *normal-form block.*

d) Interchange of statements:

Suppose a block has the following two adjacent statements:

t1 : = b + c
t2 : = x + y

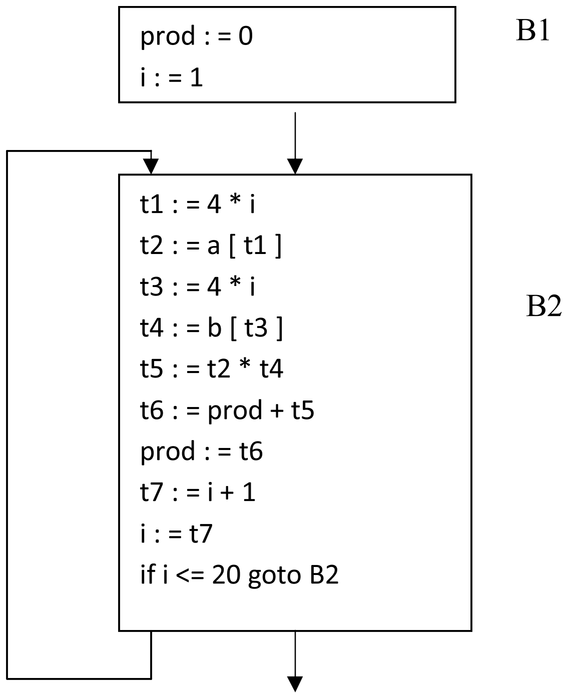
We can interchange the two statements without affecting the value of the block if and only if neither **x** nor **y** is **t_1** and neither **b** nor **c** is **t_2**.

2. Algebraic transformations:

Algebraic transformations can be used to change the set of expressions computed by a basic block into an algebraically equivalent set.
Examples:
i) x : = x + 0 or x : = x * 1 can be eliminated from a basic block without changing the set of expressions it computes.
ii) The exponential statement x : = y * * 2 can be replaced by x : = y * y.

Flow Graphs

- Flow graph is a directed graph containing the flow-of-control information for the set of basic blocks making up a program.
- The nodes of the flow graph are basic blocks. It has a distinguished initial node.
- E.g.: Flow graph for the vector dot product is given as follows:



- B_1 is the *initial* node. B_2 immediately follows B_1, so there is an edge from B_1 to B_2. The target of jump from last statement of B_1 is the first statement B_2, so there is an edge from B_1 (last statement) to B_2 (first statement).
- B_1 is the *predecessor* of B_2, and B_2 is a *successor* of B_1.

Loops

- A loop is a collection of nodes in a flow graph such that
    1. All nodes in the collection are *strongly connected*.
    2. The collection of nodes has a unique *entry*.
- A loop that contains no other loops is called an inner loop.

NEXT-USE INFORMATION

- If the name in a register is no longer needed, then we remove the name from the register and the register can be used to store some other names.

Input: Basic block B of three-address statements

Output: At each statement i: x= y op z, we attach to i the liveliness and next-uses of x, y and z.

Method: We start at the last statement of B and scan backwards.

1. Attach to statement i the information currently found in the symbol table regarding the next-use and liveliness of x, y and z.
2. In the symbol table, set x to "not live" and "no next use".
3. In the symbol table, set y and z to "live", and next-uses of y and z to i.

Symbol Table:

Names	**Liveliness**	**Next-use**
x	not live	no next-use
y	Live	i
z	Live	i

A SIMPLE CODE GENERATOR

- A code generator generates target code for a sequence of three- address statements and effectively uses registers to store operands of the statements.

- For example: consider the three-address statement **a := b+c**
  It can have the following sequence of codes:

ADD R_j, R_i Cost = 1 // if R_i contains b and R_j contains c

(or)

ADD c, R_i Cost = 2 // if c is in a memory location

(or)

MOV c, R_j Cost = 3 // move c from memory to Rj and add

ADD R_j, R_i

Register and Address Descriptors:

- A register descriptor is used to keep track of what is currently in each registers. The register descriptors show that initially all the registers are empty.
- An address descriptor stores the location where the current value of the name can be found at run time.

A code-generation algorithm:

The algorithm takes as input a sequence of three-address statements constituting a basic block. For each three-address statement of the form x : = y op z**,** perform the following actions:

1. Invoke a function *getreg* to determine the location L where the result of the computation y op z should be stored.

2. Consult the address descriptor for y to determine y', the current location of y. Prefer the register for y' if the value of y is currently both in memory and a register. If the value of y is not already in L, generate the instruction **MOV y' , L** to place a copy of y in L.

3. Generate the instruction **OP z' , L** where z' is a current location of z. Prefer a register to a memory location if z is in both. Update the address descriptor of x to indicate that x is in location L. If x is in L, update its descriptor and remove x from all other descriptors.

4. If the current values of y or z have no next uses, are not live on exit from the block, and are in registers, alter the register descriptor to indicate that, after execution of x : = y op z , those registers will no longer contain y or z.

Generating Code for Assignment Statements:

- The assignment d : = (a-b) + (a-c) + (a-c) might be translated into the following three-address code sequence:

```
t : = a – b
u : = a – c
v : = t + u
d : = v + u
```

with d live at the end.

Code sequence for the example is:

Statements	**Code Generated**	**Register descriptor**	**Address descriptor**
		Register empty	
t : = a - b	MOV a, R_0 SUB b, R0	R_0 contains t	t in R_0
u : = a - c	MOV a , R1 SUB c , R1	R_0 contains t R1 contains u	t in R_0 u in R1
v : = t + u	ADD R_1, R_0	R_0 contains v R_1 contains u	u in R_1 v in R_0
d : = v + u	ADD R_1, R_0 MOV R_0, d	R_0 contains d	d in R_0 d in R_0 and memory

Generating Code for Indexed Assignments

The table shows the code sequences generated for the indexed assignment statements
a : = b [i] and **a [i] : = b**

Statements	**Code Generated**	**Cost**
a : = b[i]	MOV b(R_i), R	2
a[i] : = b	MOV b, a(R_i)	3

Generating Code for Pointer Assignments

The table shows the code sequences generated for the pointer assignments
a : = *p and ***p : = a**

Statements	**Code Generated**	**Cost**
a : = *p	MOV *R_p, a	2
*p : = a	MOV a, *R_p	2

Generating Code for Conditional Statements

Statement	**Code**
if x < y goto z	CMP x, y CJ< z /* jump to z if condition code is negative */
x : = y +z if x < 0 goto z	MOV y, R_0 ADD z, R_0 MOV R_0,x CJ< z

THE DAG REPRESENTATION FOR BASIC BLOCKS

- A DAG for a basic block is a **directed acyclic graph** with the following labels on nodes:
  1. Leaves are labeled by unique identifiers, either variable names or constants.
  2. Interior nodes are labeled by an operator symbol.
  3. Nodes are also optionally given a sequence of identifiers for labels to store the computed values.
- DAGs are useful data structures for implementing transformations on basic blocks.
- It gives a picture of how the value computed by a statement is used in subsequent statements.
- It provides a good way of determining common sub - expressions.

Algorithm for construction of DAG

Input: A basic block

Output: A DAG for the basic block containing the following information:

1. A label for each node. For leaves, the label is an identifier. For interior nodes, an operator symbol.
2. For each node a list of attached identifiers to hold the computed values.

Case (i) x : = y OP z

Case (ii) x : = OP y

Case (iii) x : = y

Method:

Step 1: If y is undefined then create node(y).

If z is undefined, create node(z) for case(i).

Step 2: For the case(i), create a node(OP) whose left child is node(y) and right child is

node(z). (Checking for common sub expression). Let n be this node.

For case(ii), determine whether there is node(OP) with one child node(y). If not create such a node.

For case(iii), node n will be node(y).

Step 3: Delete x from the list of identifiers for node(x). Append x to the list of attached

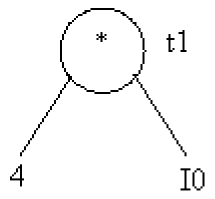
identifiers for the node n found in step 2 and set node(x) to n.

Example: Consider the block of three- address statements:

1. t_1 := 4* i
2. t_2 := a[t_1]
3. t_3 := 4* i
4. t_4 := b[t_3]
5. t_5 := t_2*t_4
6. t_6 := prod+t_5
7. prod := t_6
8. t_7 := i+1
9. i := t_7
10. if i<=20 goto (1)

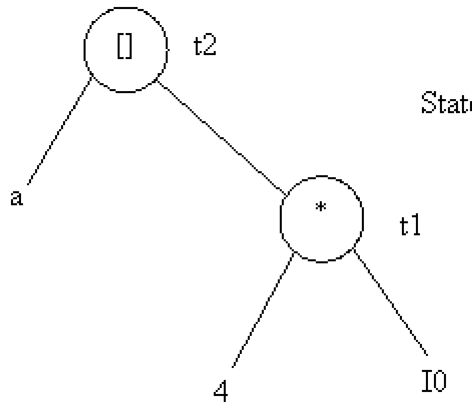
Stages in DAG Construction

(a)



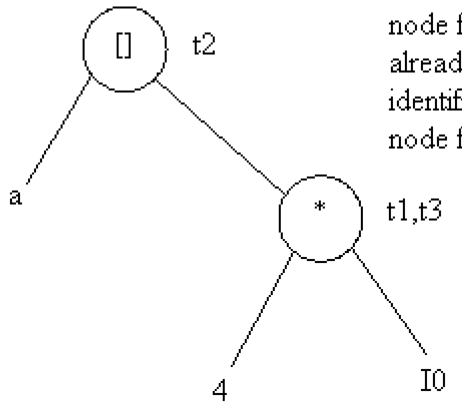
Statement (1)

(b)



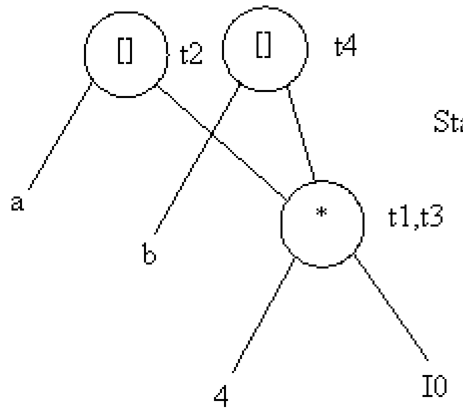
Statement (2)

(c)



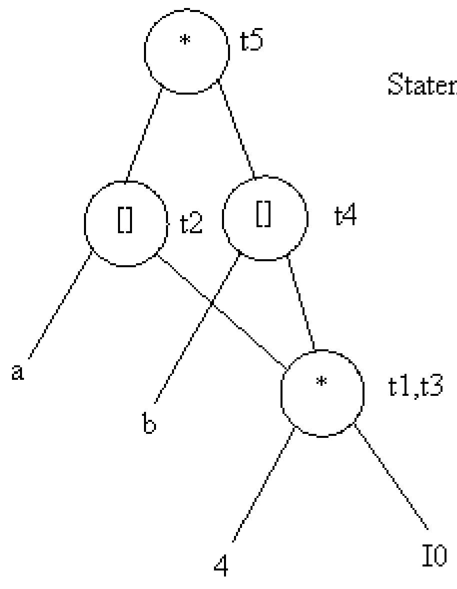
node for 4*I0 exist already, hence attach identifier t3 to the existing node for Statement (3)

(d)



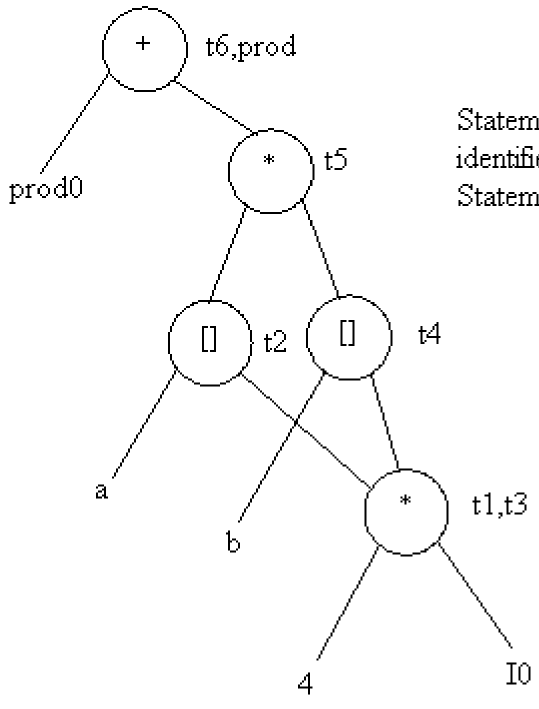
Statement (4)

(e)



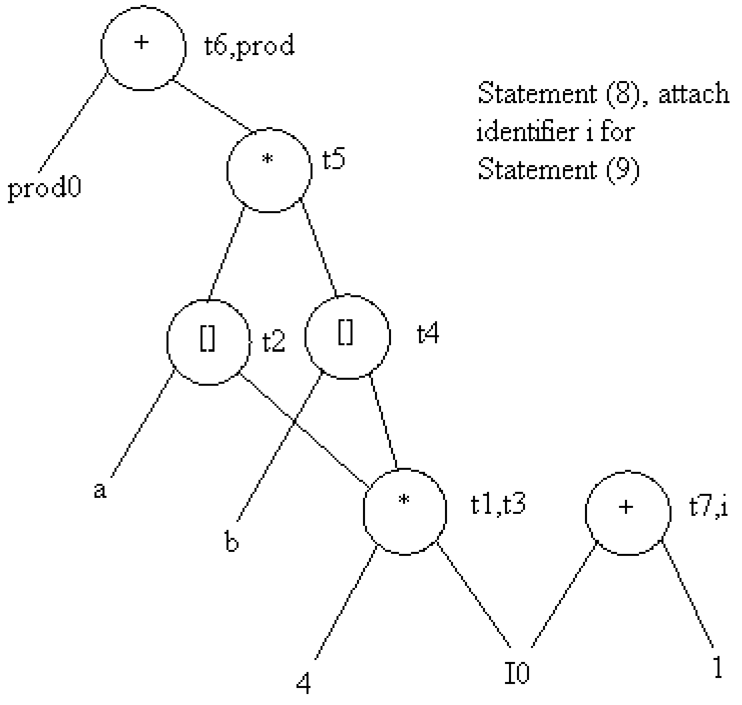
Statement (5)

(f)



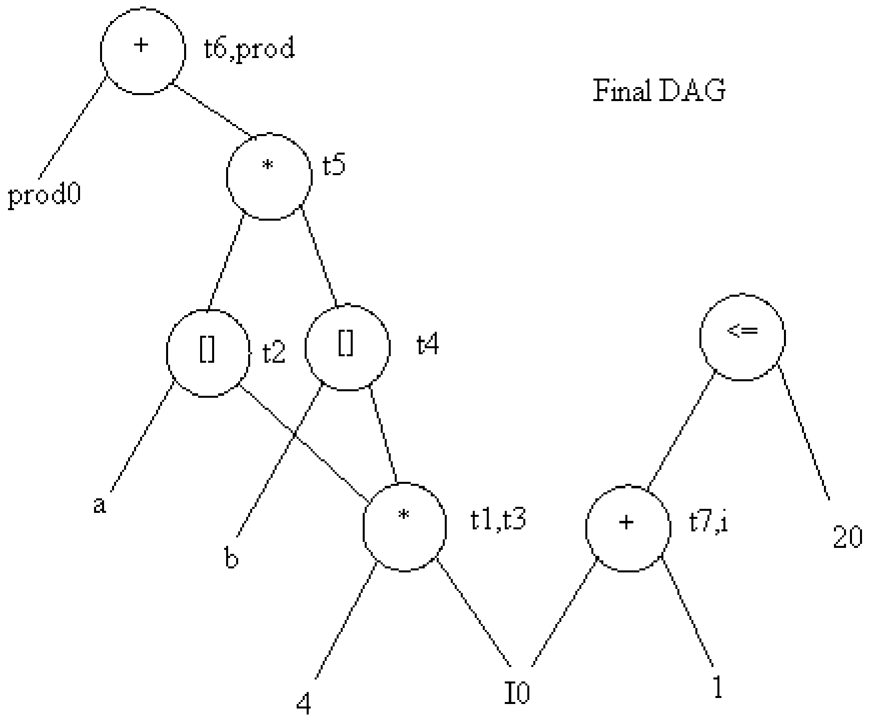
Statement (6), attach identifier prod for Statement (7)

(g)



Statement (8), attach identifier i for Statement (9)

(h)



Final DAG

Application of DAGs:

1. We can automatically detect common sub expressions.
2. We can determine which identifiers have their values used in the block.
3. We can determine which statements compute values that could be used outside the block.

GENERATING CODE FROM DAGs

The advantage of generating code for a basic block from its dag representation is that, from a dag we can easily see how to rearrange the order of the final computation sequence than we can starting from a linear sequence of three-address statements or quadruples.

Rearranging the order

The order in which computations are done can affect the cost of resulting object code.

For example, consider the following basic block:

$t_1 := a + b$
$t_2 := c + d$
$t_3 := e - t_2$
$t_4 := t_1 - t_3$

Generated code sequence for basic block:

MOV a , R_0
ADD b , R_0
MOV c , R_1
ADD d , R_1
MOV R_0 , t_1
MOV e , R_0
SUB R_1 , R_0
MOV t_1 , R_1
SUB R_0 , R_1
MOV R_1 , t_4

Rearranged basic block:
Now t1 occurs immediately before t4.

$t_2 := c + d$
$t_3 := e - t_2$
$t_1 := a + b$
$t_4 := t_1 - t_3$

Revised code sequence:

MOV c , R_0
ADD d , R_0
MOV a , R_0
SUB R_0 , R_1
MOV a , R_0
ADD b , R_0
SUB R_1 , R_0
MOV R_0 , t_4

In this order, two instructions **MOV R_0 , t_1** and **MOV t_1 , R_1** have been saved.

A Heuristic ordering for Dags

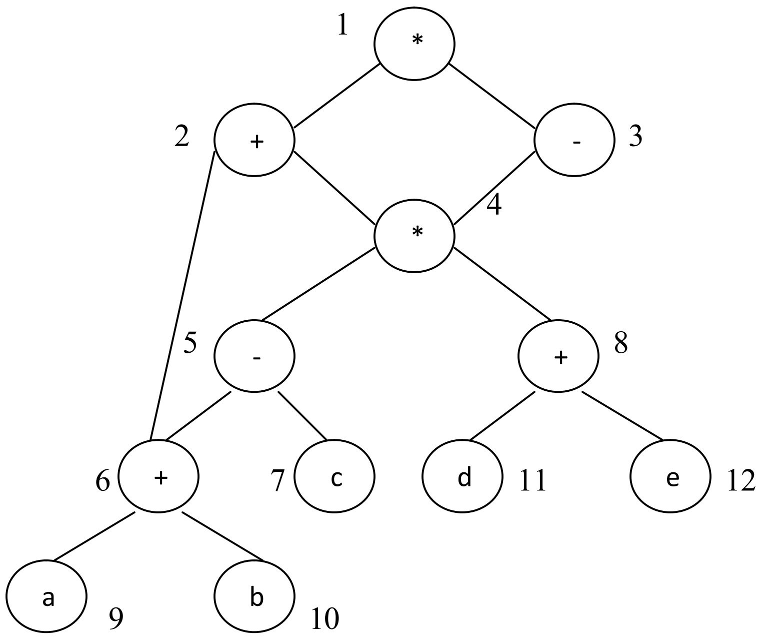
The heuristic ordering algorithm attempts to make the evaluation of a node immediately follow the evaluation of its leftmost argument.

The algorithm shown below produces the ordering in reverse.

Algorithm:

```
1) while unlisted interior nodes remain do begin
2)      select an unlisted node n, all of whose parents have been listed;
3)      list n;
4)      while the leftmost child m of n has no unlisted parents and is not a leaf do
            begin
5)              list m;
6)              n : = m
            end
        end
```

Example: Consider the DAG shown below:



Initially, the only node with no unlisted parents is 1 so set n=1 at line (2) and list 1 at line (3).

Now, the left argument of 1, which is 2, has its parents listed, so we list 2 and set n=2 at line (6).

Now, at line (4) we find the leftmost child of 2, which is 6, has an unlisted parent 5. Thus we select a new n at line (2), and node 3 is the only candidate. We list 3 and proceed down its left chain, listing 4, 5 and 6. This leaves only 8 among the interior nodes so we list that.

The resulting list is 1234568 and the order of evaluation is 8654321.

Code sequence:

t_8 : = d + e
t_6 : = a + b
t_5 : = t_6 – c
t_4 : = t_5 * t_8
t_3 : = t4 – e
t_2 : = t_6 + t_4
t_1 : = t_2 * t_3

This will yield an optimal code for the DAG on machine whatever be the number of registers.