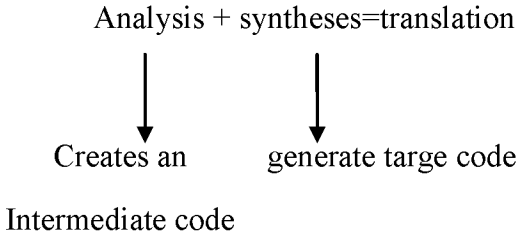
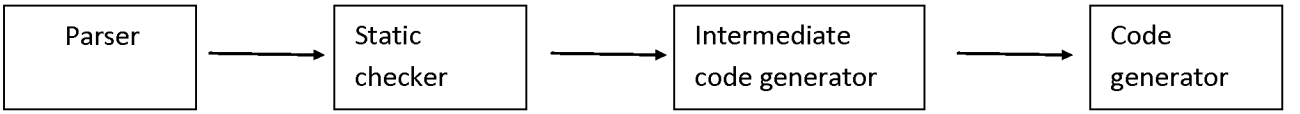
Part-A:Semantic analysis

1. Intermediate code forms:

An intermediate code form of source program is an internal form of a program created by the compiler while translating the program created by the compiler while translating the program from a high –level language to assembly code(or)object code(machine code).an intermediate source form represents a more attractive form of target code than does assembly. An optimizing Compiler performs optimizations on the intermediate source form and produces an object module.



In the analysis –synthesis model of a compiler, the front-end translates a source program into an intermediate representation from which the back-end generates target code, in many compilers the source code is translated into a language which is intermediate in complexity between a HLL and machine code .the usual intermediate code introduces symbols to stand for various temporary quantities.



position of intermediate code generator

We assume that the source program has already been parsed and statically checked.. the various intermediate code forms are:

a) Polish notation
b) Abstract syntax trees(or)syntax trees
c) Quadruples
d) Triples — three address code
e) Indirect triples
f) Abstract machine code(or)pseudocopde

a. postfix notation:

The ordinary (infix) way of writing the sum of a and b is with the operator in the middle: a+b. the postfix (or postfix polish)notation for the same expression places the operator at the right end, as ab+.

In general, if e1 and e2 are any postfix expressions, and Ø to the values denoted by e1 and e2 is indicated in postfix notation nby e1e2Ø.no parentheses are needed in postfix notation because the position and priority (number of arguments) of the operators permits only one way to decode a postfix expression.

Example:

1. (a+b)*c in postfix notation is ab+c*,since ab+ represents the infix expression(a+b).

2. a*(b+c)is abc+* in postfix.

3. (a+b)*(c+d) is ab+cd+* in postfix.

Postfix notation can be generalized to k-ary operators for any k>=1.if k-ary operator Ø is applied to postfix expression e1,e2,.....ek, then the result is denoted by e1e2.....ek Ø. if we know the priority of each operator then we can uniquely decipher any postfix expression by scanning it from either end.

Example:

Consider the postfix string ab+c*.

The right hand * says that there are two arguments to its left. since the next –to-rightmost symbol is c, simple operand, we know c must be the second operand of *.continuing to the left, we encounter the operator +.we know the sub expression ending in + makes up the first operand of *.continuing in this way ,we deduce that ab+c* is "parsed" as (((a,b)+),c)*.

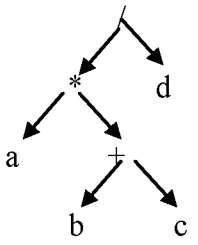
b. syntax tree:

The parse tree itself is a useful intermediate-language representation for a source program, especially in optimizing compilers where the intermediate code needs to extensively restructure.

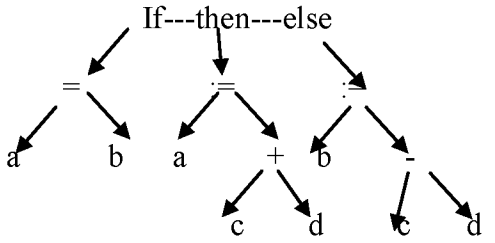
A parse tree, however, often contains redundant information which can be eliminated, Thus producing a more economical representation of the source program. One such variant of a parse tree is what is called an (abstract) syntax tree, a tree in which each leaf represents an operand and each interior node an operator.

Exmples:

1) Syntax tree for the expression a*(b+c)/d



2) syntax tree for **if a=b then a:=c+d else b:=c-d**



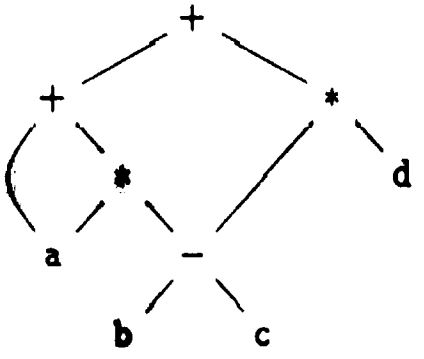
Three-Address Code:

• In three-address code, there is at most one operator on the right side of aninstruction; that is, no built-up arithmetic expressions are permitted.

x+y*z □ t1 = y * z

t2 = x + t1

• **Example**



(a) DAG

$t_1 = b - c$

$t_2 = a * t_1$

$t_3 = a + t_2$

$t_4 = t_1 * d$

$t_5 = t_3 + t_4$

(b) Three-address code

Figure 6.8: A DAG and its corresponding three-address code

Problems:
Write the 3-address code for the following expression
1. if(x + y * z > x * y +z)
a=0;
2. (2 + a * (b – c / d)) / e
3. A :=b * -c + b * -c

Address and Instructions
•
• **Example** Three-address code is built from two concepts: addresses and instructions.
• An address can be one of the following:
– A name: A source name is replaced by a pointer to its symbol table entry.
• **A name:** For convenience, allow source-program names to
Appear as addresses in three-address code. In an
Implementation, a source name is replaced by a pointer to
its symbol-table entry, where all information about the name is kept.
– A constant
• **A constant:** In practice, a compiler must deal with many different types of constants and variables
– A compiler-generated temporary
• **A compiler-generated temporary.** It is useful, especially in optimizing compilers, to create a distinct name each time a temporary is needed. These temporaries can be combined, if possible, when registers are allocated to variables.
A list of common three-address instruction forms:
Assignment statements
– x= y op z, where op is a binary operation
– x= op y, where op is a unary operation
– Copy statement: x=y
– Indexed assignments: x=y[i] and x[i]=y
– Pointer assignments: x=&y, *x=y and x=*y
Control flow statements
– Unconditional jump: goto L
– Conditional jump: if x relop y goto L ; if x goto L; if False x goto L
– Procedure calls: call procedure p with n parameters and **return y**, is
Optional
param x1
param x2
…
param xn
call p, n

– **do i = i +1; while (a[i]<v);**

```
L:   t1 = i + 1
     i = t1
     t2 = i * 8
     t3 = a [ t2 ]
     if t3 < v goto L
```

(a) Symbolic labels.

```
100: t1 = i + 1
101: i = t1
102: t2 = i * 8
103: t3 = a [ t2 ]
104: if t3 < v goto 100
```

(b) Position numbers.

The multiplication i * 8 is appropriate for an array of elements that each take 8 units of space.

C. quadruples:

• Three-address instructions can be implemented as objects or as record with fields for the operator and operands.

• Three such representations

– Quadruple, triples, and indirect triples

• A quadruple (or quad) has four fields: op, arg1, arg2, and result.

Example

D. Triples

• A triple has only three fields: op, arg1, and arg2

• Using triples, we refer to the result of an operation x op y by its position, rather by an explicit temporary name.

Example

```
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
		...		

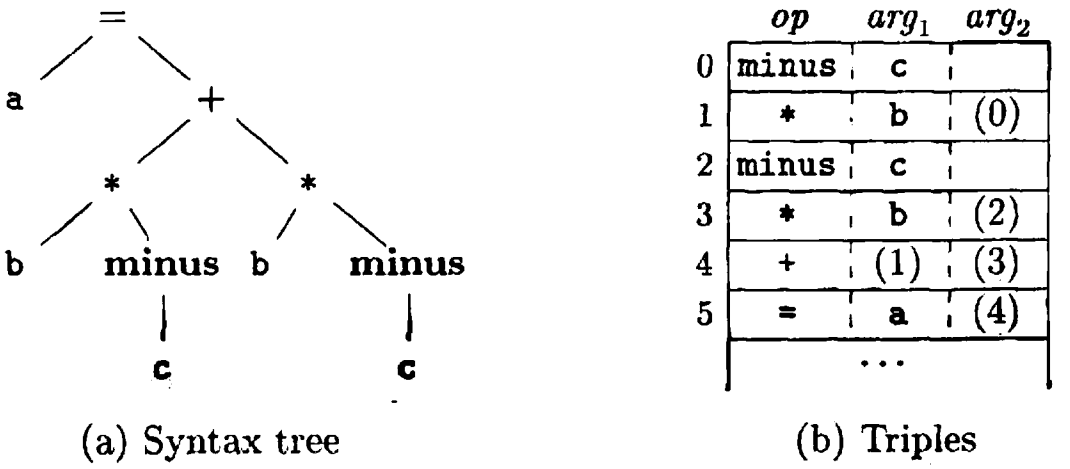
(b) Quadruples

d. Triples:

• A triple has only three fields: op, arg1, and arg2

• Using triples, we refer to the result of an operation x op y by its position, rather by an explicit temporary name.

Example



(a) Syntax tree (b) Triples

Fig: Representations of a = b * - c + **b** * - c

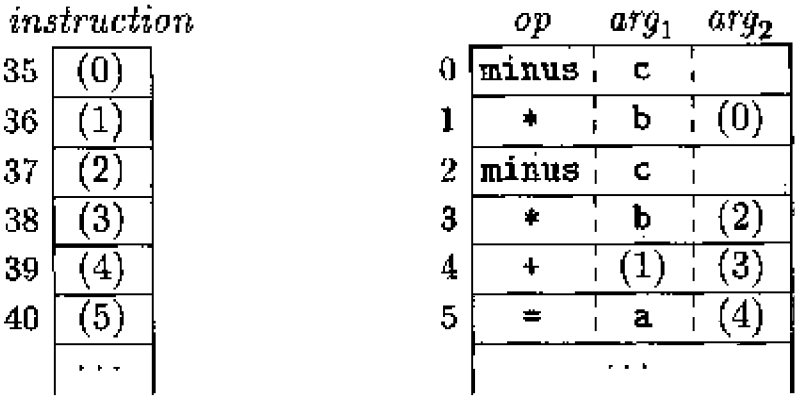


Fig: Indirect triples representation of 3-address code

-> The benefit of **Quadruples** over **Triples** can be seen in an optimizing compiler, where instructions are often moved around.

->With **quadruples**, if we move an instruction that computes a temporary **t**, then the instructions that use **t** require no change. With **triples**, the result of an operation is referred to by its position, so moving an instruction may require changing all references to that result. **This problem does not occur with indirect triples**.

Single-Assignment Static Form

Static single assignment form (SSA) is an intermediate representation that facilitates certain code optimization.

• Two distinct aspects distinguish SSA from three –address code.

– All assignments in SSA are to variables with distinct names; hence the term static single-assignment.

```
p = a + b
q = p - c
p = q * d
p = e - p
q = p + q
```

(a) Three-address code.

$$
\begin{aligned}
p_1 &= a + b \\
q_1 &= p_1 - c \\
p_2 &= q_1 * d \\
p_3 &= e - p_2 \\
q_2 &= p_3 + q_1
\end{aligned}
$$

(b) Static single-assignment form.

Figure 6.13: Intermediate program in three-address code and SSA

```
if (flag) x = -1; else x = 1;
y = x * a;
```

$$
\begin{aligned}
&\text{if (flag) } x_1 = -1; \text{ else } x_2 = 1; \\
&x_3 = \phi(x_1, x_2)
\end{aligned}
$$

2. **Type Checking:**

•A compiler has to do semantic checks in addition to syntactic checks.

•Semantic Checks

–Static –done during compilation

–Dynamic –done during run-time

•**Type checking** is one of these static checking operations.

–we may not do all type checking at compile-time.

–Some systems also use dynamic type checking too.

•A **type system** is a collection of rules for assigning type expressions to the parts of a program.

•A **type checker** implements a type system.

•A **sound type** system eliminates run-time type checking for type errors.

•A programming language is strongly-typed, if every program its compiler accepts will execute without type errors.

In practice, some of type checking operations is done at run-time (so, most of the programming languages are not strongly yped).

–Ex: int x[100]; … x[i] □ most of the compilers cannot guarantee that i will be between 0 and 99

Type Expression:

•The type of a language construct is denoted by a type expression.

•A type expression can be:

–A basic type

•a primitive data type such as integer, real, char, Boolean, …

•type-error to signal a type error

•void: no type

–A type name

•a name can be used to denote a type expression.

–A type constructor applies to other type expressions.

•**arrays**: If T is a type expression, then array (I,T)is a type expression where I denotes index range. Ex: array (0..99,int)

•**products**: If T1and T2 are type expressions, then their Cartesian product T1 x T2 is a type expression. Ex: int x int

•**pointers**: If T is a type expression, then pointer (T) is a type expression. Ex: pointer (int)

•**functions**: We may treat functions in a programming language as mapping from a domain type D to a range type R. So, the type of a function can be denoted by the type expression D→R where D are R type expressions. Ex: int→int represents the type of a function which takes an int value as parameter, and its return type is also int.

Type Checking of Statements:

S ->d= E { if (id.type=E.type then S.type=void

else S.type=type-error }

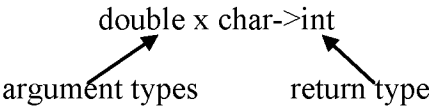
S ->if E then S1	{ if (E.type=boolean then S.type=S1.type else S.type=type-error }
S->while E do S1	{ if (E.type=boolean then S.type=S1.type else S.type=type-error }

Type Checking of Functions:

E->E1(E2)	{ if (E2.type=s and E1.type=s□ t) thenE.type=t else E.type=type-error }

Ex: int f(double x, char y) { ... }

f: double x char->int



Structural Equivalence of Type Expressions:

•How do we know that two type expressions are equal?

•As long as type expressions are built from basic types (no type names), we may use structural equivalence between two type expressions

Structural Equivalence Algorithm (sequin):

if (s and t are same basic types) then return true

else if (s=array(s1,s2) and t=array(t1,t2)) then return (sequiv(s1,t1) and sequiv(s2,t2))

else if (s = s1 x s2and t = t1 x t2) then return (sequiv(s1,t1) and sequiv(s2,t2))

else if (s=pointer(s1) and t=pointer(t1)) then return (sequiv(s1,t1))

else if (s = s1 □ s2and t = t1 □ t2) then return (sequiv(s1,t1) and sequiv(s2,t2))

else return false

Names for Type Expressions:

•In some programming languages, we give a name to a type expression, and we use that name as a type expression afterwards.

type link = ↑cell; ? p,q,r,s have same types ?

var p,q : link;

var r,s : ↑cell

•How do we treat type names?

–Get equivalent type expression for a type name (then use structural equivalence), or

–Treat a type name as a basic type

3. Syntax Directed Translation:

- A formalist called as syntax directed definition is used fort specifying translations for programming language constructs.
- A syntax directed definition is a generalization of a context free grammar in which each grammar symbol has associated set of attributes and each and each productions is associated with a set of semantic rules

Definition of (syntax Directed definition) SDD :

SDD is a generalization of CFG in which each grammar productions X->α is associated with it a set of semantic rules of the form

a: = f(b1,b2…..bk)

Where a is an attributes obtained from the function f.

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

– Each grammar symbol is associated with a set of attributes.

– This set of attributes for a grammar symbol is partitioned into two subsets called **synthesized** and **inherited** attributes of that grammar symbol.

– Each production rule is associated with a set of semantic rules.

• Semantic rules set up dependencies between attributes which can be represented by a dependency graph.

• This dependency graph determines the evaluation order of these semantic rules.

• Evaluation of a semantic rule defines the value of an attribute. But a semantic rule may also have some side effects such as printing a value.

The two attributes for non terminal are :

1) Synthesized attribute (S-attribute) : (↑)

An attribute is said to be synthesized attribute if its value at a parse tree node is determined from attribute values at the children of the node

2) Inherited attribute: (↑,→)

An inherited attribute is one whose value at parse tree node is determined in terms of attributes at the parent and | or siblings of that node.

- The attribute can be string, a number, a type, a, memory location or anything else.
- The parse tree showing the value of attributes at each node is called an annotated parse tree.

The process of computing the attribute values at the node is called annotating or decorating the parse tree.**Terminals** can have synthesized attributes, but not inherited attributes.

Annotated Parse Tree

• A parse tree showing the values of attributes at each node is called an **Annotated parse tree**.

• The process of computing the attributes values at the nodes is called **annotating** (or **decorating**) of the parse tree.

• Of course, the order of these computations depends on the dependency graph induced by the semantic rules.
Ex1:1) Synthesized Attributes :
Ex: Consider the CFG :
S→ EN
E→ E+T
E→E-T
E→ T
T→ T*F
T→T/F
T→F
F→ (E)
F→digit
N→;

Solution: The syntax directed definition can be written for the above grammar by using semantic actions for each production.

Production rule	**Semantic actions**
S →EN	S.val=E.val
E →E1+T	E.val =E1.val + T.val
E →E1-T	E.val = E1.val – T.val
E →T	E.val =T.val
T →T*F	T.val = T.val * F.val
T →T|F	T.val =T.val | F.val
F → (E)	F.val =E.val
T →F	T.val =F.val
F →digit	F.val =digit.lexval
N →;	can be ignored by lexical Analyzer as; I is terminating symbol

For the Non-terminals E,T and F the values can be obtained using the attribute "Val".

The taken digit has synthesized attribute "lexval".

In S→EN, symbol S is the start symbol. This rule is to print the final answer of expressed.

Following steps are followed to Compute S attributed definition

1. Write the SDD using the appropriate semantic actions for corresponding production rule of the given Grammar.

2. The annotated parse tree is generated and attribute values are computed. The Computation is done in bottom up manner.

3. The value obtained at the node is supposed to be final output.

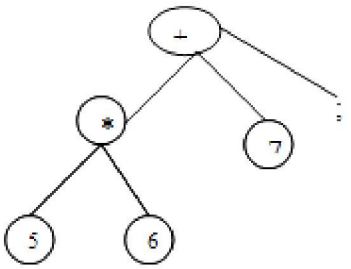
PROBLEM 1:

Consider the string 5*6+7; Construct Syntax tree, parse tree and annotated tree.

Solution:

The corresponding annotated parse tree is shown below for the string 5*6+7;

Syntax tree:



Annotated parse tree :

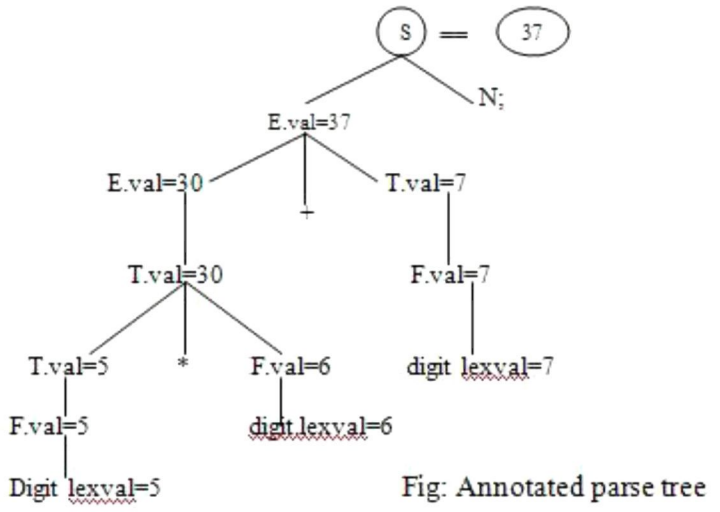


Fig: Annotated parse tree

Advantages: SDDs are more readable and hence useful for specifications

Disadvantages: not very efficient.

Ex2:

PROBLEM : Consider the grammar that is used for Simple desk calculator. Obtain the Semantic action and also the annotated parse tree for the string

3*5+4n.

L→En

E→E1+T

E→T

T→T1*F

T→F

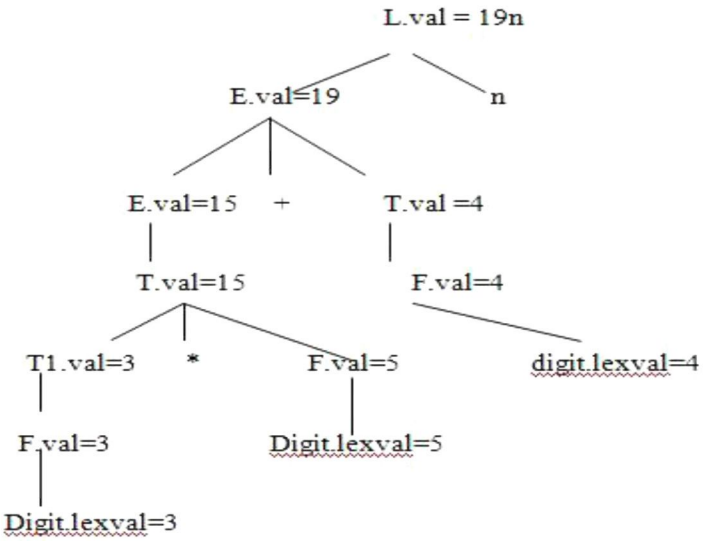
F→ (E)

F→digit

Solution :

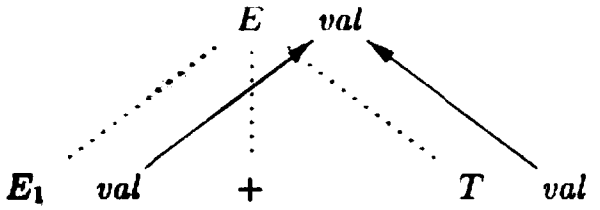
Production rule	**Semantic actions**
L→En	L.val=E.val
E→E1+T	E.val=E1.val + T.val
E→T	E.val=T.val
T→T1*F	T.val=T1.val*F.val
T→F	T.val=F.val
F→(E)	F.val=E.val
F→digit	F.val=digit.lexval

The corresponding annotated parse tree U shown below, for the string 3*5+4n.



Dependency Graphs:

Figure 5.6. *E.val* is synthesized from E_1.*val* and E_2.*val*



Dependency graph and topological sort:

- For each parse-tree node, say a node labeled by grammar symbol X, the dependency graph has a node for each attribute associated with X.
- If a semantic rule associated with a production p defines the value of synthesized attribute A.b in terms of the value of X.c. Then the dependency graph has an edge from X.c to A.b
- If a semantic rule associated with a production p defines the value of inherited attribute B.c in terms of the value X.a. Then , the dependency graph has an edge from X.a to B.c.

Applications of Syntax-Directed Translation

• Construction of syntax Trees

– The nodes of the syntax tree are represented by objects with a suitable number of fields.

– Each object will have an ***op*** field that is the label of the node.

– The objects will have additional fields as follows

• If the node is a leaf, an additional field holds the lexical value for the leaf. A constructor function ***Leaf (op, val)*** creates a leaf object.

• If nodes are viewed as records, the Leaf returns a pointer to a new record for a leaf.

· If the node is an interior node, there are as many additional fields as the node has children in the syntax tree. A constructor function

Node takes two or more arguments:

Node (op , c1,c2,…..ck) creates an object with first field ***op*** and ***k*** additional fields for the ***k*** children ***c1,c2,…..ck***

Syntax-Directed Translation Schemes

A SDT scheme is a context-free grammar with program fragments embedded within production bodies .The program fragments are called semantic actions and can appear at any position within the production body.

Any SDT can be implemented by first building a parse tree and then pre-forming the actions in a left-to-right depth first order. i.e during preorder traversal.

The use of SDT's to implement two important classes of SDD's

1. If the grammar is LR parsable, then SDD is S-attributed.
2. If the grammar is LL parsable, then SDD is L-attributed.

Postfix Translation Schemes

The postfix SDT implements the desk calculator SDD with one change: the action for the first production prints the value. As the grammar is LR, and the SDD is S-attributed.

L →E n {print(E.val);}
E → E1 + T { E.val = E1.val + T.val }
E → E1 - T { E.val = E1.val - T.val }
E → T { E.val = T.val }
T → T1 * F { T.val = T1.val * F.val }
T → F { T.val = F.val }
F → (E) { F.val = E.val }
F → digit { F.val = digit.lexval }

	PRODUCTION	SEMANTIC RULES
1)	$E \rightarrow E_1 + T$	$E.node = \textbf{new}\ Node('+', E_1.node, T.node)$
2)	$E \rightarrow E_1 - T$	$E.node = \textbf{new}\ Node('-', E_1.node, T.node)$
3)	$E \rightarrow T$	$E.node = T.node$
4)	$T \rightarrow (E)$	$T.node = E.node$
5)	$T \rightarrow \textbf{id}$	$T.node = \textbf{new}\ Leaf(\textbf{id}, \textbf{id}.entry)$
6)	$T \rightarrow \textbf{num}$	$T.node = \textbf{new}\ Leaf(\textbf{num}, \textbf{num}.val)$

PART-B

Symbol tables

Symbol table:

A **symbol table** is a major data structure used in a compiler:

- Associates **attributes** with identifiers used in a program.
- For instance, a **type attribute** is usually associated with each identifier.
- A symbol table is a necessary component.
- Definition (declaration) of identifiers appears once in a program
- Use of identifiers may appear in many places of the program text
- Identifiers and attributes are entered by the analysis phases
- When processing a definition (declaration) of an identifier
- In simple languages with only global variables and implicit declarations:
- The scanner can enter an identifier into a symbol table if it is not already there
- In block-structured languages with scopes and explicit declarations:
- The parser and/or semantic analyzer enter identifiers and corresponding attributes
- Symbol table information is used by the analysis and synthesis phases
- To verify that used identifiers have been defined (declared)
- To verify that expressions and assignments are semantically correct – **type checking**
- To generate intermediate or target code

Symbol Table Interface:

The basic operations defined on a symbol table include:

- **allocate** – to allocate a new empty symbol table
- **free** – to remove all entries and free the storage of a symbol table
- **insert** – to insert a name in a symbol table and return a pointer to its entry
- **lookup** – to search for a name and return a pointer to its entry
- **set_attribute** – to associate an attribute with a given entry
- **get_attribute** – to get an attribute associated with a given entry
- Other operations can be added depending on requirement

  For example, a **delete** operation removes a name previously inserted

  Some identifiers become invisible (out of scope) after exiting a block

- This interface provides an abstract view of a symbol table.
- Supports the simultaneous existence of multiple tables
  - Implementation can vary without modifying the interface

Basic Implementation Techniques:

First consideration is how to **insert** and **lookup** names

Variety of implementation techniques

Unordered List

Simplest to implement

Implemented as an array or a linked list

Linked list can grow dynamically – alleviates problem of a fixed size array

Insertion is fast $O(1)$, but lookup is slow for large tables – $O(n)$ on average

Ordered List

If an array is sorted, it can be searched using binary search – $O(\log_2 n)$

Insertion into a sorted array is expensive – $O(n)$ on average

Useful when set of names is known in advance – table of reserved words

Binary Search Tree

Can grow dynamically

Insertion and lookup are $O(\log_2 n)$ on average

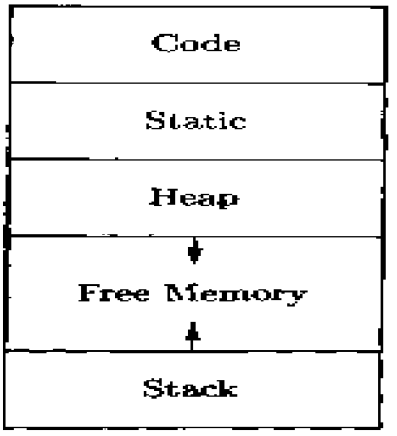
Hash Tables and Hash Functions:

- A **hash table** is an array with index range: 0 to *TableSize* – 1
- Most commonly used data structure to implement symbol tables
- Insertion and lookup can be made very fast – $O(1)$
- A **hash function** maps an identifier name into a table index
  - A hash function, h(name), should depend solely on name
  - h(name) should be computed quickly
  - h should be **uniform** and **randomizing** in distributing names
  - All table indices should be mapped with equal probability
  - Similar names should not cluster to the same table index.

Storage Allocation:

- Compiler must do the storage allocation and provide access to variables and data
- Memory management
    - Stack allocation
    - Heap management
    - Garbage collection

Storage Organization:



• Assumes a logical address space

- Operating system will later map it to physical addresses, decide how touse cache memory, etc.

• Memory typically divided into areas for

- Program code
- Other static data storage, including global constants and compilergenerated data
- Stack to support call/return policy for procedures
- Heap to store data that can outlive a call to a procedure

Static vs. Dynamic Allocation:

- Static: Compile time, Dynamic: Runtime allocation
- Many compilers use some combination of following
- Stack storage: for local variables, parameters and so on
- Heap storage: Data that may outlive the call to the procedure that created it

- Stack allocation is a valid allocation for procedures since procedure calls are nest

Example:

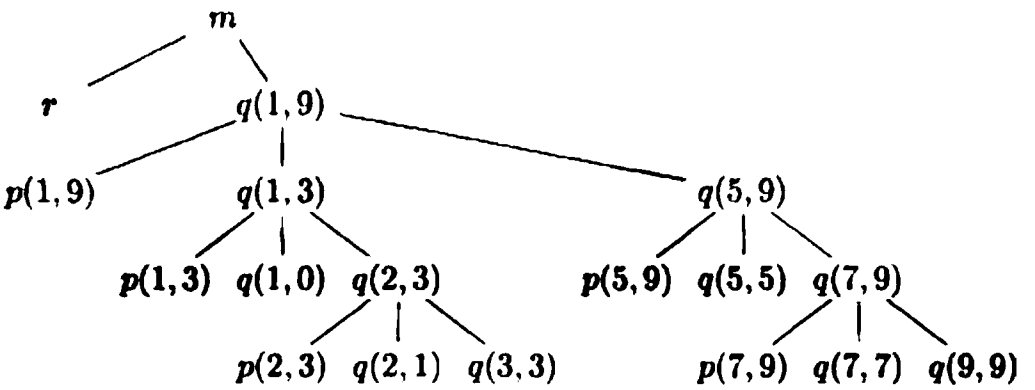
Consider the quick sort program

```
int a[11];
void readArray() { /* Reads 9 integers into a[1], ..., a[9]. */
    int i;
    ...
}
int partition(int m, int n) {
    /* Picks a separator value v, and partitions a[m..n] so that
       a[m..p-1] are less than v, a[p] = v, and a[p+1..n] are
       equal to or greater than v. Returns p. */
    ...
}
void quicksort(int m, int n) {
    int i;
    if (n > m) {
        i = partition(m, n);
        quicksort(m, i-1);
        quicksort(i+1, n);
    }
}
main() {
    readArray();
    a[0] = -9999;
    a[10] = 9999;
    quicksort(1,9);
}
```

Activation for Quicksort:

```
enter main()
    enter readArray()
    leave readArray()
    enter quicksort(1,9)
        enter partition(1,9)
        leave partition(1,9)
        enter quicksort(1,3)
            ...
        leave quicksort(1,3)
        enter quicksort(5,9)
            ...
        leave quicksort(5,9)
    leave quicksort(1,9)
leave main()
```

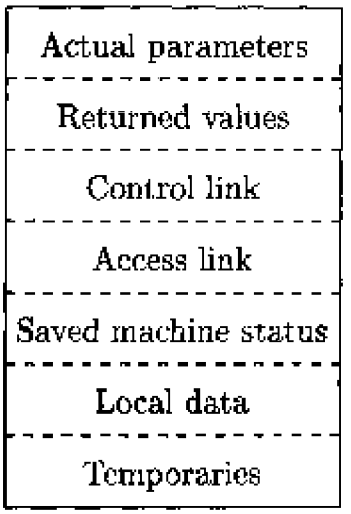
Activation tree representing calls during an execution of quicksort:



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 (sometimes called a frame)
- The root of activation tree is at the bottom of the stack
- The current execution path specifies the content of the stack with the last
- Activation has record in the top of the stack.

A General Activation Record



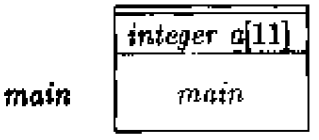
Activation Record

- Temporary values
- Local data
- A saved machine status
- An "access link"
- A control link

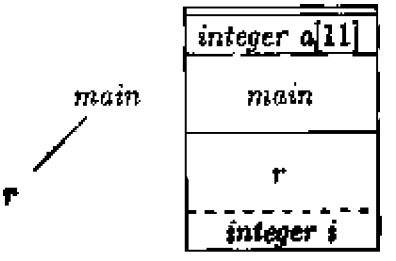
- Space for the return value of the called function
- The actual parameters used by the calling procedure

- Elements in the activation record:
- Temporary values that could not fit into registers.
- Local variables of the procedure.
- Saved machine status for point at which this procedure called. Includes return address and contents of registers to be restored.
- Access link to activation record of previous block or procedure in lexical scope chain.
- Control link pointing to the activation record of the caller.
- Space for the return value of the function, if any.
- actual parameters (or they may be placed in registers, if possible)

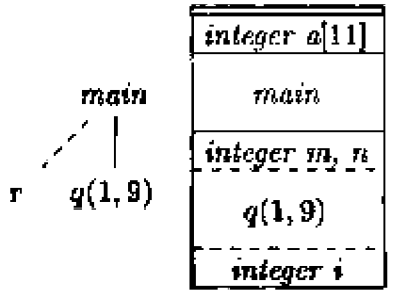
Downward-growing stack of activation records:

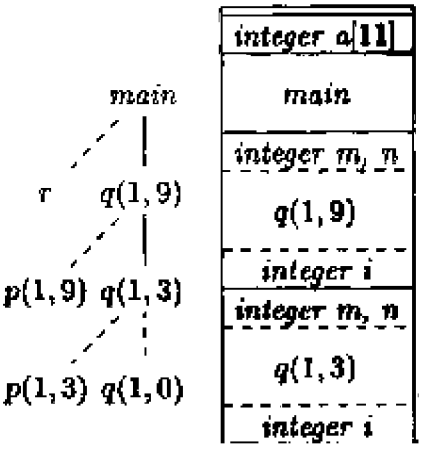


(a) Frame for *main*



(b) r is activated

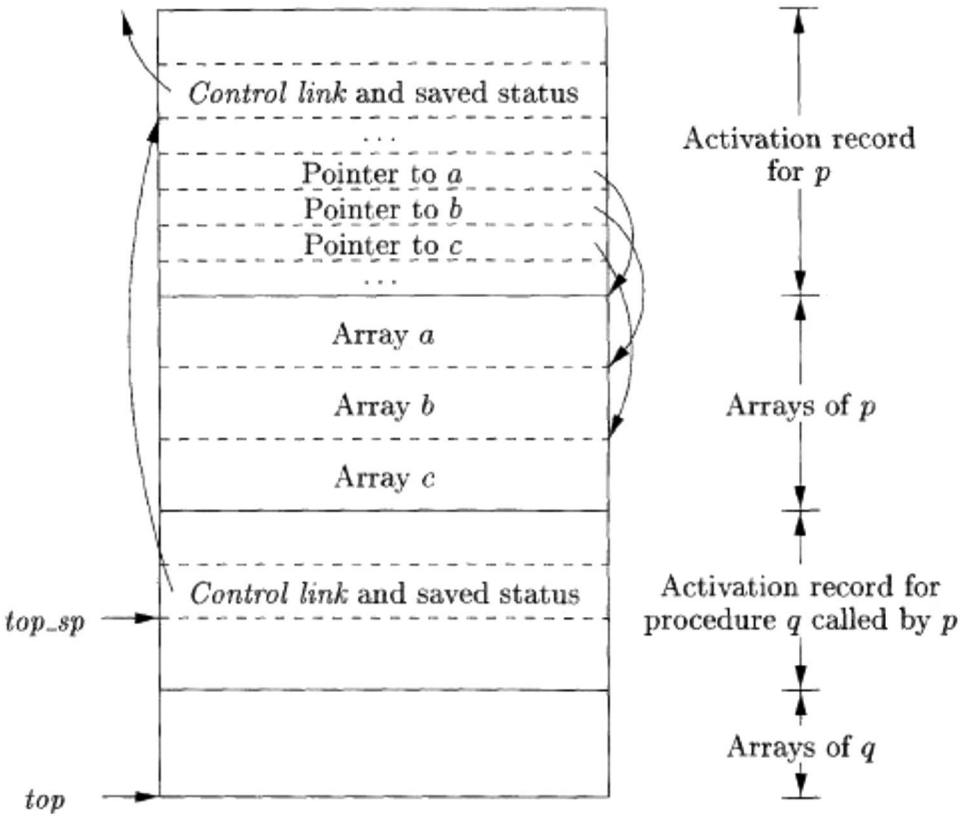




Designing Calling Sequences:

- Values communicated between caller and callee are generally placed at the beginning of callee's activation record
- Fixed-length items: are generally placed at the middle
- Items whose size may not be known early enough: are placed at the end of activation record
- We must locate the top-of-stack pointer judiciously: a common approach is to have it point to the end of fixed length fields

Access to dynamically allocated arrays:



ML:

- ML is a functional language
- Variables are defined, and have their unchangeable values initialized, by a statementof the form:
  val (name) = (expression)
- Functions are defined using the syntax:

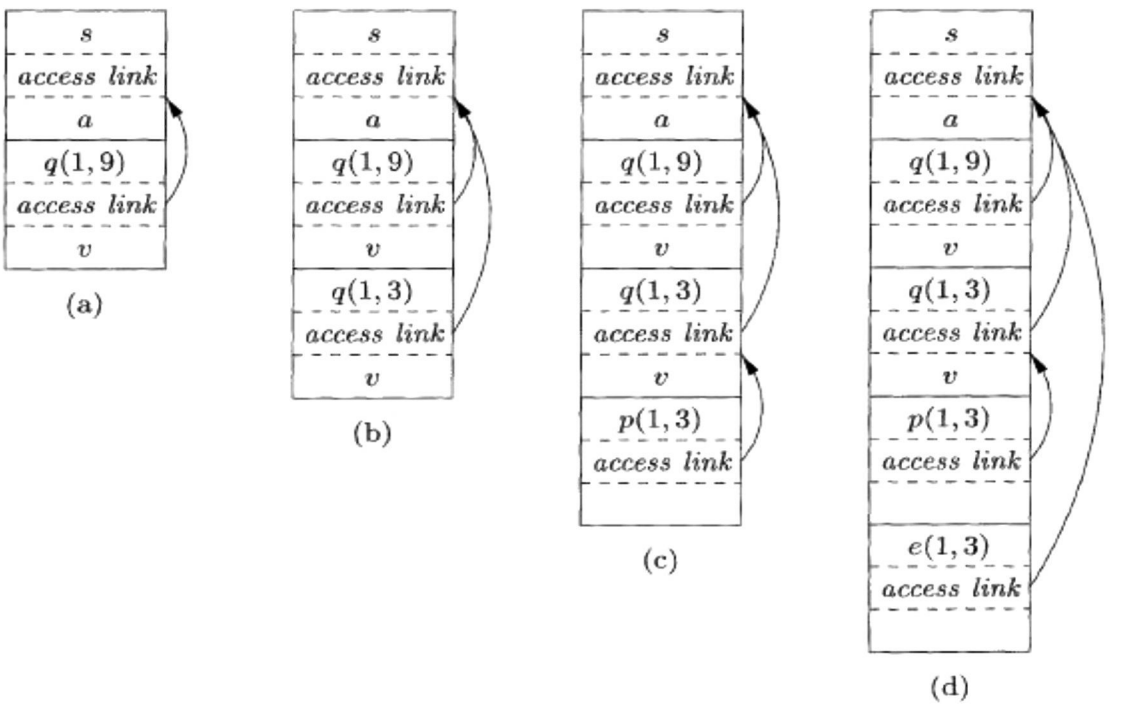
fun (name) ((arguments)) = (body)

- For function bodies we shall use let-statements of the form:
  let (list of definitions) in (statements) end

A version of quick sort, in ML style, using nested functions:

```
1) fun sort(inputFile, outputFile) =
       let
2)         val a = array(11,0);
3)         fun readArray(inputFile) = ··· ;
4)               ··· a ··· ;
5)         fun exchange(i,j) =
6)               ··· a ··· ;
7)         fun quicksort(m,n) =
               let
8)                 val v = ··· ;
9)                 fun partition(y,z) =
10)                      ··· a ··· v ··· exchange ···
               in
11)                ··· a ··· v ··· partition ··· quicksort
               end
       in
12)        ··· a ··· readArray ··· quicksort ···
       end;
```

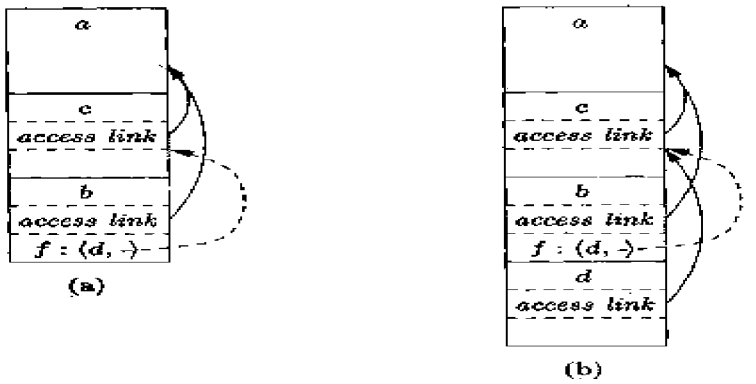
Access links for finding nonlocal data:



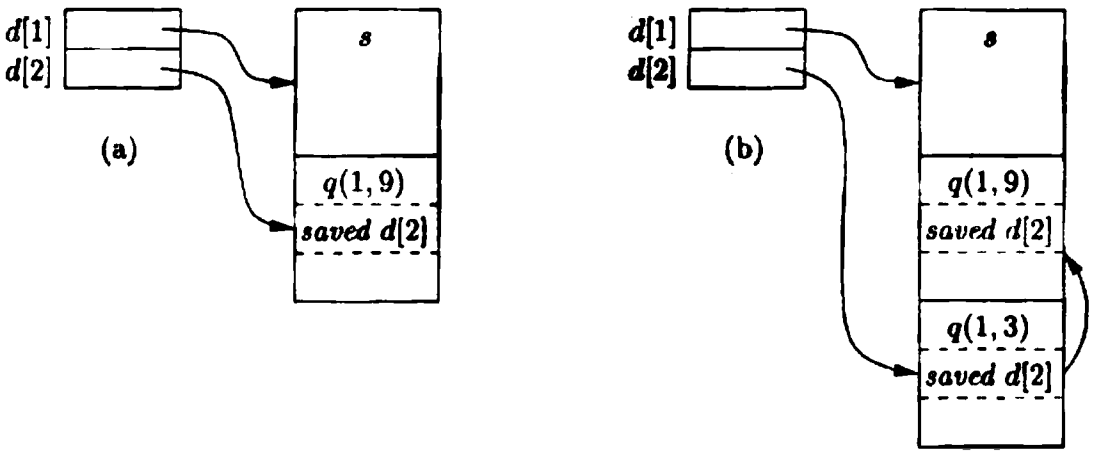
Sketch of ML program that uses function-parameters:

```
fun a(x) =
    let
        fun b(f) =
            ··· f ··· ;
        fun c(y) =
            let
                fun d(z) = ···
            in
                ··· b(d) ···
            end
    in
        ··· c(1) ···
    end;
```

Actual parameters carry their access link with them:



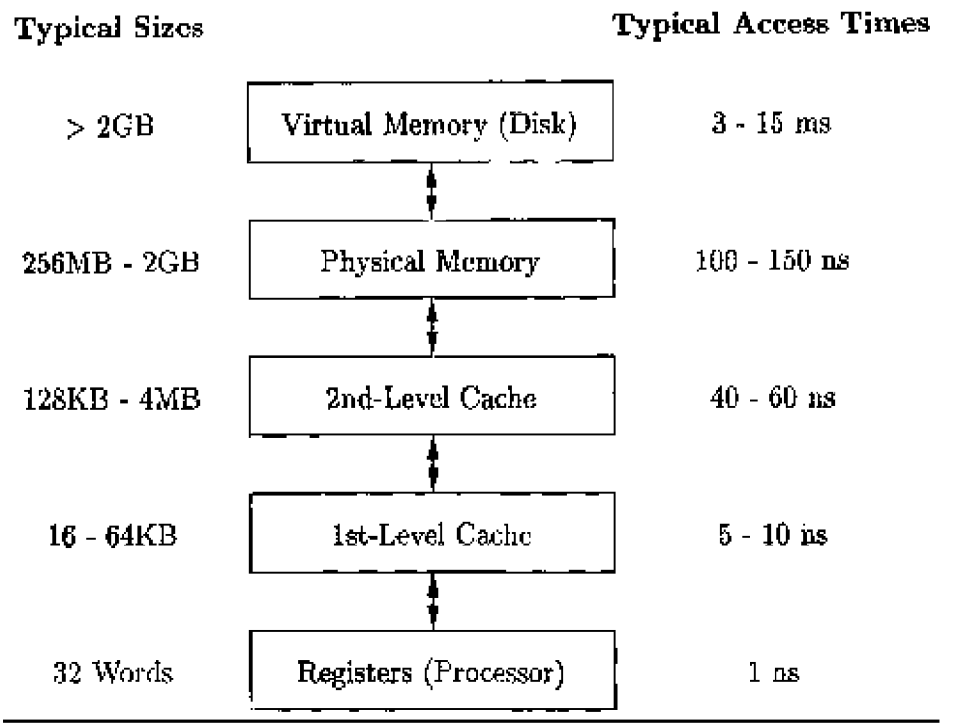
Maintaining the Display:



Memory Manager:

- Two basic functions:
- Allocation
- Deallocation
- Properties of memory managers:
- Space efficiency
- Program efficiency
- Low overhead

Typical Memory Hierarchy Configurations:



Locality in Programs:

The conventional wisdom is that programs spend 90% of their time executing 10% of the code:

- Programs often contain many instructions that are never executed.
- Only a small fraction of the code that could be invoked is actually executed in atypical run of the program.
- The typical program spends most of its time executing innermost loops and tight recursive cycles in a program.