UNIT-III

TYPE CHECKING & RUN TIME ENVIRONMENT

TYPE CHECKING -

(1) A compiler must check that the source program follows with the syntactic and semantic conventions of the source language. This checking is called static checking. Examples of static checks include - Type checks, Flow-of-control checks, Uniqueness Check, Name-related checks.

A compiler should report an error if an operator is applied to an incompatible operand. This checking is called Type Checking.

token stream → PARSER → syntax tree → TYPE CHECKER → syntax tree → INTERMEDIATE CODE GENERATOR → intermediate representation

Position of type checker

(2) Type Systems -

Type Expression - Type of a language construct. It is either a basic type or is formed by applying an operator called a type constructor to other type expressions.

A type system is a collection of rules for assigning type expressions to the various parts of a program. A type checker implements a type system. Different type systems may be used by different compilers or processors of the same language.

Checking done by a compiler is said to be static checking of types, while checking done when the target program runs is termed dynamic checking of types.

A sound type system eliminates the need for dynamic checking for type errors because it allows us to determine statically that these errors cannot occur when the target program runs.

Type checker should have a property of Error Recovery

Date ___/___/___
Page ______

③ <u>Specification of simple type checker</u> -

The type checker is a translation scheme that synthesizes the type of each expression from the types of its subexpressions.

Consider a language that requires that an identifier be declared with a type before the variable is used. For simplicity, we will declare all identifiers before using them in a single expression -

$P \rightarrow D ; E$

$D \rightarrow D ; D \mid id : T$

$T \rightarrow char \mid integer \mid array\ [num]\ of\ T \mid \uparrow T$

$E \rightarrow literal \mid num \mid id \mid E\ mod\ E \mid E[E] \mid E\uparrow$

→ The part of a translation scheme that saves the <u>type of an identifier</u> -

$P \rightarrow D ; E$

$D \rightarrow D ; D$

$D \rightarrow id : T$ { addtype (id.entry, T.type) }

$T \rightarrow char$ { T.type := char }

$T \rightarrow integer$ { T.type := integer }

$T \rightarrow array\ [num]\ of\ T_1$ { T.type := array (1.. num.val, T_1.type) }

$T \rightarrow \uparrow T_1$ { T.type := pointer (T_1.type) }

→ <u>Type checking of expressions</u> -

$E \rightarrow literal$ { E.type := char }

$E \rightarrow num$ { E.type := integer }

$E \rightarrow id$ { E.type := lookup (id.entry) }

$E \rightarrow E_1\ mod\ E_2$ { E.type := if E_1.type = integer and E_2.type = integer then integer else type_error }

$E \rightarrow E_1 [E_2]$ { E.type := if E_2.type = integer and E_1.type = array(s, t) then t else type_error }

$E \rightarrow E_1 \uparrow$ { E.type := if E_1.type = pointer (t) then t else type_error }

Type checking of statements -

Translation scheme for checking the type of statements -

$S \rightarrow id := E$ { S.type := if id.type = E.type then void else type_error }

$S \rightarrow$ if E then S_1 { S.type := if E.type = boolean then S_1.type else type_error }

$S \rightarrow$ while E do S_1 { S.type := if E.type = boolean then S_1.type else type_error }

$S \rightarrow S_1 ; S_2$ { S.type := if S_1.type = void and S_2.type = void then void else type_error }

Type checking of functions -

$E \rightarrow E_1(E_2)$ { E.type := if E_2.type = s and E_1.type = $s \rightarrow t$ then t else type_error }

$T \rightarrow T_1 ' \rightarrow ' T_2$ { T.type := T_1.type $\rightarrow$ T_2.type }

(4) Equivalence of type expressions -

→ Structural Equivalence of type expressions -

Two expressions are either the same basic type, or are formed by applying the same constructor to structurally equivalent types. That is, two type expressions are structurally equivalent if and only if they are identical. Testing structural equivalence of two type expressions s and t -

```
function sequiv(s, t): boolean;
begin
    if s and t are the same basic type then
        return true
    else if s = array(s1, s2) and t = array(t1, t2) then      //array
        return sequiv(s1, t1) and sequiv(s2, t2)
    else if s = s1 x s2 and t = t1 x t2 then                    //products
        return sequiv(s1, t1) and sequiv(s2, t2)
    else if s = pointer(s1) and t = pointer(t1) then            //pointers
        return sequiv(s1, t1)
    else if s = s1 -> s2 and t = t1 -> t2 then                  //functions
        return sequiv(s1, t1) and sequiv(s2, t2)
    else return false
end
```

Date ___/___/___

Page ________

→ Name Equivalence for Type Expressions -

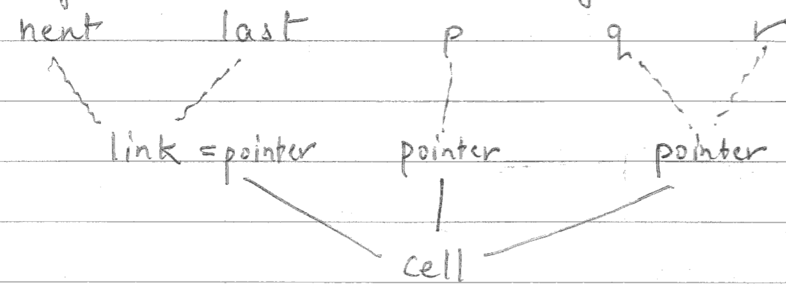
Name equivalence views each type name as a distinct type so two type expressions are name equivalent if and only if they are identical.

Eg -
```
type link = ↑cell;
var next : link;
    last : link;
    p    : ↑cell;
    q, r : ↑cell;
```

Structural equivalence

VARIABLE	TYPE EXPRESSION	
next	link	Name equivalence
last	link	
p	pointer (cell)	
q	pointer (cell)	Name equivalence
r	pointer (cell)	

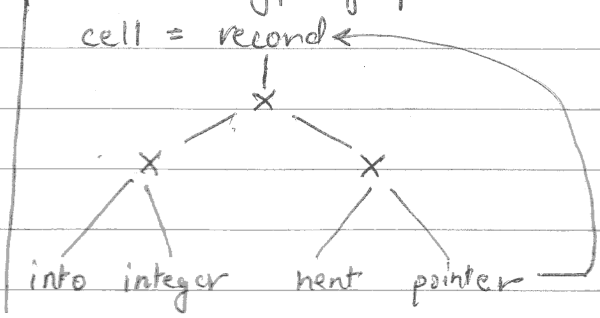
Type graph of above example is given as -



→ Cycles in representation of types -

Basic data structures like linked lists and trees are often defined recursively. Recursively defined type names can be substituted out if we are willing to introduce cycles into the type graph.

Eg -
```
type link = ↑cell;
     cell = record
              info : integer;
              next : link
            end;
```



⑤ Type conversions -

Conversion from one type to another is said to be <u>implicit</u> if it is to be done automatically by the compiler. Implicit type conversions, also called as <u>coercions</u>, are limited in many languages to situations where no information is lost in principle.

Date ___/___/___

Page ______

Conversion is said to be explicit if the programmer must write something to cause the conversion.

Type checking rules for coercion from integer to real is given as -

PRODUCTION	SEMANTIC RULES
$E \rightarrow num$	E.type := integer
$E \rightarrow num.num$	E.type := real
$E \rightarrow id$	E.type := lookup(id.entry)
$E \rightarrow E_1 \; op \; E_2$	E.type := if E_1.type = integer and E_2.type = integer then integer else if E_1.type = integer and E_2.type = real then real else if E_1.type = real and E_2.type = integer then real else if E_1.type = real and E_2.type = real then real else type_error

(6) Overloading of functions and operators -

The resolution of overloading is sometimes referred to as operator identification, because it determines which operation an operator symbol denotes.

→ Set of possible Types for a subexpression -

Instead of a single type, a subexpression standing alone may have a set of possible types.

Determining the set of possible types of an expression is given as -

PRODUCTION	SEMANTIC RULE
$E' \rightarrow E$	E'.types := E.types
$E \rightarrow id$	E.types := lookup(id.entry)
$E \rightarrow E_1(E_2)$	E.types := { t | there exists an s in E_2.types such that $s \rightarrow t$ is in E_1.types }

→ Narrowing the set of possible types -

Given a unique type from the context, we can narrow down the type choices for each subexpression. If this process does not result in a unique type for each subexpression, then a type error is declared for the expression.

Date ___/___/___
Page ______

PRODUCTION	SEMANTIC RULES
$E' = E$	$E'.types := E.types$ $E.unique := \text{if } E'.types = \{t\} \text{ then } t \text{ else type\_error}$ $E'.code := E.code$
$E \rightarrow id$	$E.types := lookup(id.entry)$ $E.code := gen(id.lexeme \text{ ':' } E.unique)$
$E \rightarrow E_1(E_2)$	$E.types := \{ s' \mid \text{there exists an } s \text{ in } E_2.types \text{ such that } s \rightarrow s' \text{ is in } E_1.types \}$ $t := E.unique$ $S := \{ s \mid s \in E_2.types \text{ and } s \rightarrow t \in E_1.types \}$ $E_2.unique := \text{if } S = \{s\} \text{ then } s \text{ else type\_error}$ $E_1.unique := \text{if } S = \{s\} \text{ then } s \rightarrow t \text{ else type\_error}$ $E.code := E_1.code \,\|\, E_2.code \,\|\, gen(\text{'apply' ':' } E.unique)$

code is used to generate PASCAL ~~code~~ program code
unique is used to find the unique type of the expression

⑦ Polymorphic functions –

A function with arguments of different types is known as polymorphic functions. To deal with polymorphisms, we extend our set of type expressions to include expression with type variables.

→ Why Polymorphic functions?

Polymorphic functions are attractive because they facilitate the implementation of algorithms that manipulate data structures, regardless of the types of the elements in the data structure.

→ Type Variables –

A type variable represents the type of an undeclared identifier. An important application of type variables is checking consistent usage of identifiers in a language that does not require identifiers to be declared before they are used.

Type inference is the problem of determining the types of a language construct from the way it is used. The term is often

applied to the problem of inferring the type of a function from its body
Eg - function deref (p)
begin
return p↑
end;
let β is a type variable of P. From p↑ ie. p must be a pointer of unknown type lets say, α then,

$\beta = pointer(\alpha)$

Furthermore, the expression p↑ has type α, so we can write the type expression for any type α,

$pointer(\alpha) \rightarrow \alpha$ for the type of the function deref.

→ A language with polymorphic functions -

A type expression with a symbol ∀ (for any type) in it will be referred to informally as a polymorphic type.

Grammer for language with polymorphic functions ~~are~~ is given as-

$P \rightarrow D; E$

$D \rightarrow D; D \mid id : Q$

$Q \rightarrow \forall$ type_variable . $Q \mid T$

$T \rightarrow T \text{ '}\rightarrow\text{' } T \mid T \times T \mid$ unary_constructor $(T) \mid$ basic_type $\mid$ type_variable $\mid (T)$

$E \rightarrow E(E) \mid E, E \mid id$.

→ The differences from the rules for ordinary functions are-

(1) Distinct occurences of a polymorphic function in the same expression need not have arguments of the same type.

(2) Since variables can appear in type expressions, we have to reexamine the notion of equivalence of types.

(3) We need a mechanism for recording the effect of unifying two expressions. In general, a type variable may appear in several type expressions.

Date ___/___/___
Page ______

Substitutions, Instances and Unification -

Information about the types represented by variables is formalized by defining a mapping from type variables to type expressions called a substitution.

```
function subst (t: type_expression): type_expression;
begin
    if t is a basic type then return t
    else if t is a variable then return S(t)
    else if t is t1 → t2 then return subst(t1) → subst(t2)
end
```

The result ~~type expression~~, $S(t)$ is called an instance of t.

Two type expressions t_1 and t_2 unify if there exists some substitution S such that $S(t_1) = S(t_2)$. Most precisely, the most general unifier of expression t_1 and t_2 is a substitution S with the following properties -

(1) $S(t_1) = S(t_2)$ and

(2) for any other substitution S' such that $S'(t_1) = S'(t_2)$, the substitution S' is an instance of S (that is, for any t, $S'(t)$ is an instance of $S(t)$).

Checking Polymorphic functions -

The rules for checking expressions generated by the grammar will be written in terms of the following operations on a graph representation of types -

(1) fresh(t) replaces the bound variables in type expression t by fresh variables and returns a pointer to a node representing the resulting type expression. Any $\forall$ symbols in t are removed in the process.

(2) unify(m,n) unifies the type expressions represented by the nodes pointed to by m and n. It has the side effect of keeping track of the substitution that makes the expressions equivalent. If the expressions fail to unify, the entire type-checking process fails.

Translation scheme for checking polymorphic functions is given as -

$E \rightarrow E_1 (E_2)$ { p := mkleaf (newtypevar);
unify (E_1.type, mknode ('→', E_2.type, p));
E.type := p }

$E \rightarrow E_1 , E_2$ { E.type := mknode ('X', E_1.type, E_2.type) }

$E \rightarrow id$ { E.type := fresh (id.type) }

RUN-TIME ENVIRONMENTS -

① Storage Organization -

The organization of run-time storage in this section can be used for languages such as Fortran, Pascal and C.

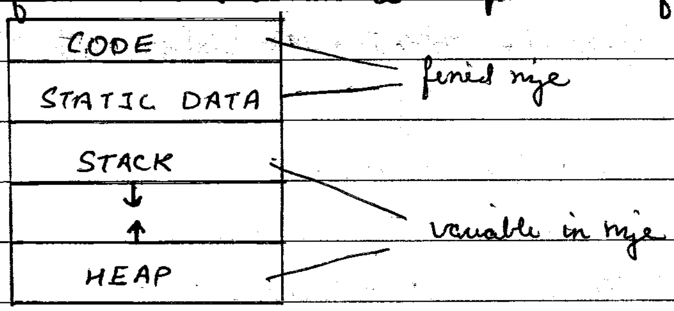
→ **Sub-division of Run-time Memory -**

The Run-time storage might be subdivided to hold -

(1) the generated target code

(2) data objects, and

(3) a counterpart of the control stack to keep track of procedure activations



Implementations of languages like Pascal and C use extensions of the control stack to manage activations of procedures.

A separate area of run-time memory, called a heap, holds all other information. Implementations of languages in which the lifetimes of activations cannot be represented by an activation tree might use the heap to keep information about activations.

→ **Activation Records -**

Information needed by a single execution of a procedure is managed using a contiguous block of storage called an activation record or frame, consisting of the collection of field which is given as -

A GENERAL ACTIVATION RECORD

RETURNED VALUE
ACTUAL PARAMETERS
OPTIONAL CONTROL LINK
OPTIONAL ACESS LINK
SAVED MACHINE STATUS
LOCAL DATA
TEMPORARIES

(1) The field for the returned value is used by the called procedure to return a value to the calling procedure.

(2) The field for actual parameters is used by the calling procedure to supply parameters to the called procedure.

(3) The optional control link points to the activation record of the caller.

(4) The optional access link is used to refer to nonlocal data held in other activation records

(5) The field for saved machine status holds information about the state of machine just before the procedure is called.

(6) The field for local data holds data that is local to an execution of a procedure

(7) The field for temporaries holds temporary values, such as those arising in the evaluation of expressions.

→ Compile time layout of local data -

The field for local data is laid out as the declarations in a procedure are examined at compile time. Variable length data is kept outside this field. We keep a count of memory locations that have been allocated for previous declarations. From the count we determine a relative address of the storage for a local with respect to some position such as the beginning of the activation record. The relative address, or offset the difference between the addresses of the position and the data object

The storage layout for data objects is strongly influenced by the addressing constraints of the target machine.

Eg - a array of 10 characters → compiler allocate 12 bytes (+ 2 bytes for padding
↳ unused.

my COMPANION

Date __/__/__
Page ____

(2) Storage allocation strategies -

A different storage-allocation strategy is used in each of the three data areas -

(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 for a data area known as a heap.

→ Static allocation -

In static allocation, names are bound to storage as the program compiled, so there is no need for a run-time support package. Since the bindings do not change at run time, every time a procedure is activated, its name are bound to the same storage locations.

However, some limitations go along with using static allocation alone are-

(1) The size of a data object and constraints on its position in memory must be known at compile time.
(2) Recursive procedures are restricted, because all activations of a procedure use the same bindings for local names.
(3) Data structures cannot be created dynamically, since there is no mechanism for storage allocation at run time.

→ Stack Allocation -

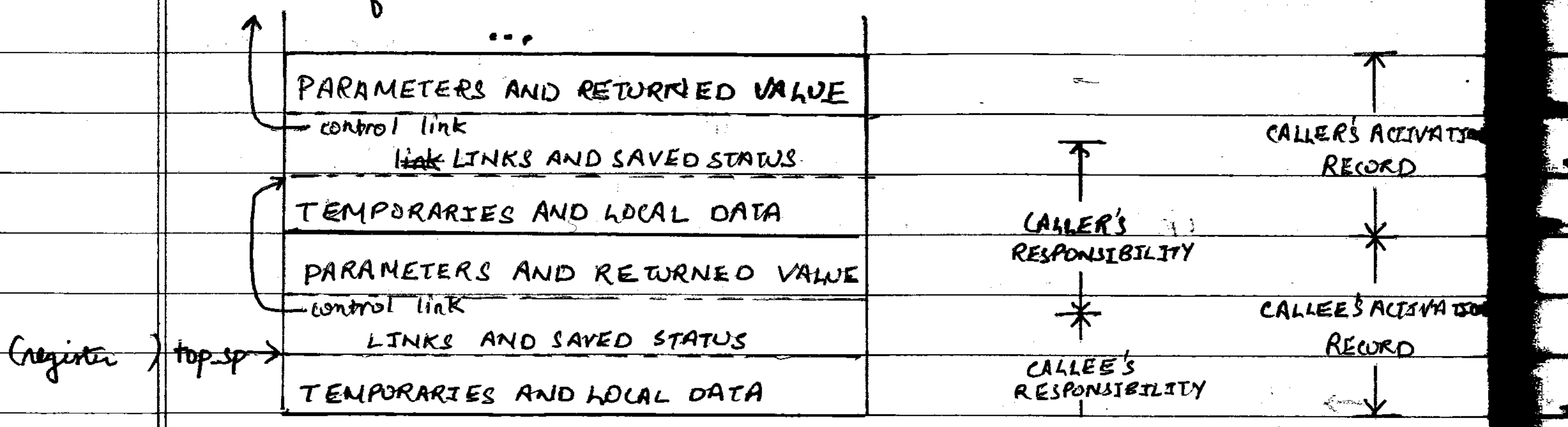
It is based on the idea of a control stack. Storage is organized as stack, and activation records are pushed and popped as activation begin and end, respectively.

→ Calling Sequences -

A call sequence allocates an activation record and enter information into its fields. A return sequence restores the state of the machine so the calling procedure can continue execution.

Calling sequences and activation records differ, even for implementation of the same language. The code in a calling sequence is often divided between the calling procedure (the caller) and the procedure it calls (the callee).

Division of tasks between caller and callee -



The call sequence is -

(1) The caller evaluate actuals

(2) The caller stores a return address and the old value of top-sp into the callee's activation record. The caller then increments top-sp to the position shown above. That is, top-sp is moved past the caller's local data and temporaries and the callee's parameter and status fields.

(3) The callee initializes its local data and begin execution.

A possible return sequence is -

(1) The callee places a return value next to the activation record of calle

(2) Using the information in the status field, the callee restores top-sp & other registers and branches to a return address in the caller's code.

(3) Although top-sp has been decremented, the caller can copy the returned value into its own activation record and use it to evaluate an expression.

→ Variable length data -

It can be handled using pointer in the activation record which points the variable length data which is not part of the activation record.

→ Dangling References -

It occurs when there is a reference to storage that has been deallocated.

Eg -
```
main() {
    int *p;
    p = dangle();
}
```

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

p is a dangling reference

Date ___/___/___

Page ______

~~Heap allocation~~ - Limitation of stack allocation. We cannot use stack if -
(1) The values of local names must be retained when an activation ends.
(2) A called activation outlives the caller.

→ Heap allocation -

To remove above limitations of stack allocation, heap allocation is used in which it parcels out pieces of contiguous storage, as needed for activation records or other objects. Pieces may be deallocated in any order, so over time the heap will consist of alternate areas that are free and in use.

For efficiency reasons, it may be helpful to handle small activation records or records of a predictable size as a special case, as follows -
(1) For each size of interest, keep a linked list of free blocks of that size.
(2) If possible, fill a request for size s with a block of size s', where s' is the smallest size greater than or equal to s. When the block is eventually deallocated, it is returned to the linked list it came from.
(3) For large blocks of storage use the heap manager.

(3) Parameter Passing -

→ Call-by-Value -

It can be implemented as follows-
(1) A formal parameter is treated just like a local name, so the storage for the formals is in the activation record of the called procedure.
(2) The caller evaluates the actual parameters and places their r-values in the storage for the formals.

→ Call-by-reference -

It can be implemented as follows-
(1) If an actual parameter is a name or an expression having an l-value, then that l-value itself is passed.
(2) However, if the actual parameter is an expression (like a+2 or 2), that has no l-value, then the expression is evaluated in a new location, and the address of that location is passed.

Date __/__/__

Page ______

→ Copy-Restore -

A hybrid between call-by-value and call-by-reference is copy-restore linkage, (also known as copy-in copy-out, or value-result). It implemented as-

(1) Before control flows to the called procedure, the actual parameters are evaluated. The r-values of the actuals are passed to the called procedure as in call-by-value. In addition, however, the l-values of those actual parameters having l-values are determined before the call.

(2) When control returns, the current r-values of the formal parameters are copied back into the l-values of the actuals, using the l-values computed before the call. Only actuals having l-values are copied, of course.

Eg -
```
program copyout (input, output);
  var a: integer;
  procedure unsafe (var x: integer);
    begin x:= 2; a:= 0 end;
  begin
    a:= 1; unsafe (a); writeln (a)
  end.
```

OUTPUT = 2 (not 0)

→ Call-by-Name -

It is traditionally defined by the copy-rule of Algol, which is -

(1) The procedure is treated as if it were a macro; that is, its body is substituted for the call in the caller, with the actual parameters literally substituted for the formals. Such a literal substitution is called macro-expansion or in-line expansion

(2) The local names of the called procedure are kept distinct from the names of the calling procedure. We can think of each local of the called procedure being systematically renamed into a distinct new name before the macro-expansion is done.

(3) The actual parameters are surrounded by parenthesis if necessary to preserve their integrity.

The usual implementation of call-by-name is to pass to the called procedure parameterless subroutines, commonly called thunks, that can evaluate the l-value or r-value of the actual parameter.

(4) Dynamic Storage allocation -

→ Explicit Allocation of fixed-sized blocks -

The simplest form of dynamic allocation involves blocks of a fixed size by linking the blocks in a list (have a pointer to point to the next block)

→ Explicit Allocation of variable-sized blocks -

When blocks are allocated and deallocated, storage can become fragmented. One method for allocating variable-sized blocks is called the first fit method.

When a block of size s is allocated, we search for the first free block that is of size $f \geq s$. This block is subdivided into a used block of size s, and a free block of size f-s.

When a block is deallocated, we check to see if it is next to a free block. If possible, the deallocated block is combined with a free block next to it to create a larger free block.

Note that allocation incurs a time overhead because we must search for a free block that is large enough. Also, combining adjacent free blocks into a larger free block prevents further fragmentation from occurring.

→ Implicit Deallocation -

It requires cooperation between the user program and the run-time package, because the latter needs to know when a storage blocks is no longer in use. This cooperation is implemented by fixing the format of storage blocks -

OPTIONAL BLOCK SIZE
OPTIONAL REFERENCE COUNT
OPTIONAL MARK
POINTERS TO BLOCKS
USER INFORMATION

FORMAT OF A BLOCK

Date __/__/__
Page ______

Two approaches can be used for implicit deallocation. They are -

(1) Reference Counts -

We keep track of the number of blocks that points directly to the present block. If this count ever drops to 0, then the block can be deallocated because it cannot be referred to.

(2) Marking Techniques -

An alternative approach is to suspend temporarily execution of the user program and use the frozen pointers to determine which blocks are in use. This approach requires all the pointers into the heap to be known.

**** The process compaction moves all used blocks to one end of the heap, so that all the free storage can be ~~allocated~~ collected into one large free block.

(5) Symbol Table -

A compiler uses a symbol table to keep track of scope and binding information about names. The symbol is searched every time a name is encountered in the source text. Changes to the table occur if a new name or new information about an existing name is discovered.

→ Symbol-Table Entries -

Each entry in the symbol table is for the declaration of a name. The format of entries does not have to be uniform, because the information saved about a name depends on the usage of the name.

→ Characters in a Name -

If there is a modest upper bound on the length of a name, then the characters in the name can be stored in the symbol-table entry as fixed size space within a record.

~~If there is no limit on the length of a name~~

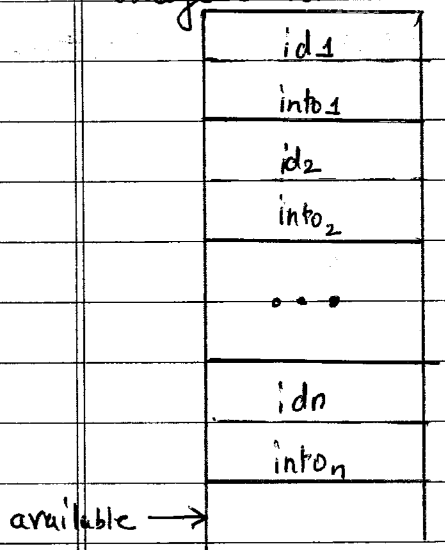
If there is no limit on the length of a name or if the limit is rarely reached, we can use a pointer with points a separate array where the name is stored.

→ Storage Allocation Information - Information about the storage locations that will be bound to names at run time is kept in the symbol table.

Symbol table mechanisms –

(1) Linear list –

The simplest and easiest to implement data structure for a symbol table is a linear list of records. We use a single array, or equivalently several arrays to store names and their associated information.

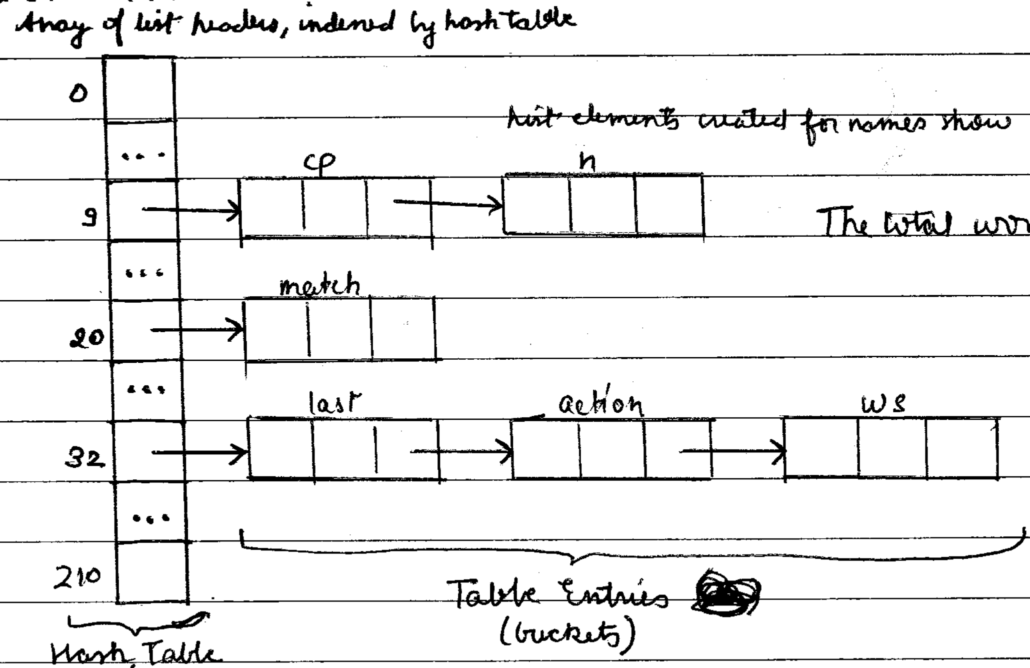


The total work for inserting n names and making e inquiries is at most $\boxed{cn(n+e)}$, where c is a constant representing the time necessary for a few machine operations.

A LINEAR LIST OF RECORDS

(2) Hash Table –

Variations of the searching technique known as hashing have been implemented in many compilers. Here we use open Hashing, where "open" refers to the property that there need ~~to~~ be no limit on the number of entries that can be made in the table.



The total work = $\boxed{n(n+e)/m}$

Date ___/___/___

Page ___

→ Representing Scope Information -

The scope rules of the source language determine which declaration is appropriate. A simple approach is to maintain a separate symbol table for each scope. In effect, the symbol table for a procedure or scope is the compile-time equivalent of an activation record.

In Hash table, we can use a scope link that chains all entries in the same scope.