Program : **B.Tech**

Subject Name: **Compiler Design**

Subject Code: **CS-603**

Semester: **6th**

UNIT- III:
Type checking: type system, specification of simple type checker, equivalence of expression, types, type conversion, overloading of functions and operations, polymorphic functions. Run time Environment: storage organization, Storage allocation strategies, parameter passing, dynamic storage allocation, Symbol table, Error Detection & Recovery, Ad-Hoc and Systematic Methods.

1. Type Checking & Run Time Environment Type Checking

Parsing cannot detect some errors. Some errors are captured during compile time called static checking (e.g., type compatibility). Languages like C, C++, C#, Java, and Haskell uses static checking. Static checking is even called early binding. During static checking programming errors are caught early. This causes program execution to be efficient. Static checking not only increases the efficiency and reliability of the compiled program, but also makes execution faster.

Type checking is not only limited to compile time, it is even performed at execution time. This is done with the help of information gathered by a compiler; the information is gathered during compilation of the source program.
Errors that are captured during run time are called dynamic checks (e.g., array bounds check or null pointers dereference check). Languages like Perl, python, and Lisp use dynamic checking. Dynamic checking is also called late binding. Dynamic checking allows some constructs that are rejected during static checking. 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 the type checking operations is done at run-time (so, most of the programming languages are not strongly typed).
For example, int x[100]; … x[i] → most of the compilers cannot guarantee that i will be between 0 and 99
A semantic analyzer mainly performs static checking. Static checks can be any one of the following type of checks:
Uniqueness checks: This ensures uniqueness of variables/objects in situations where it is required. For example, in most of the languages no identifier can be used for two different definitions in the same scope.
Flow of control checks: Statements that cause flow of control to leave a construct should have a place to transfer flow of control. If this place is missing, it is confusion. For example, in C language, “break” causes flow of control to exit from the innermost loop. If it is used without a loop, it confuses where to leave the flow of control.
Type checks: A compiler should report an error if an operator is applied to incompatible operands. For example, for binary addition, operands are array and a function is incompatible. In a function, the number of arguments should match with the number of formals and the corresponding types.
Name-related checks: Sometimes, the same name must appear two or more times. For example, in ADA, a loop or a block may have a name that appears at the beginning and end of the construct. The compiler must check whether the same name is used at both places.
What does semantic analysis do? It performs checks of many kinds which may include

- All identifiers are declared before being used.
- Type compatibility.
- Inheritance relationships.
- Classes defined only once.
- Methods in a class defined only once.
- Reserved words are not misused.

In this chapter we focus on type checking. The above examples indicate that most of the other static checks are routine and can be implemented using the techniques of SDT discussed in the previous chapter. Some of them can be combined with other activities. For example, for uniqueness check, while entering the

identifier into the symbol table, we can ensure that it is entered only once. Now let us see how to design a type checker.
A type checker verifies that the type of a construct matches with that expected by its context. For example, in C language, the type checker must verify that the operator “%” should have two integer operands dereferencing is applied through a pointer, indexing is done only on an array, a user-defined function is applied with correct number and type of arguments. The goal of a type checker is to ensure that operations are applied to the correct type of operands. Type information collected by a type checker is used later by code generator.

1.1 Type Systems
Consider the assembly language program fragment. Add R1, R2, and R3. What are the types of operands R1, R2, R3? Based on the possible type of operands and its values, operations are legal. It doesn’t make sense to add a character and a function pointer in C language. It does make sense to add two float or int values. Irrespective of the type, the assembly language implementation remains the same for add. A language’s type system specifies which operations are valid for which types. A type system is a collection of rules for assigning types to the various parts of a program. A type checker implements a type system. Types are represented by type expressions. Type system has a set of rules defined that take care of extracting the data types of each variables and check for the compatibility during the operation.

1.2 Type Expressions
The type expressions are used to represent the type of a programming language construct. Type expression can be a basic type or formed by recursively applying an operator called a type constructor to other type expressions. The basic types and constructors depend on the source language to be verified. Let us define type expression as follows:

- A basic type is a type expression
- Boolean, char, integer, real, void, type_error
- A type constructor applied to type expressions is a type expression
- Array: array(I, T)
- Array (I,T) is a type expression denoting the type of an array with elements of type T and index set I, where T is a type expression. Index set I often represents a range of integers. For example, the Pascal declaration

    var C: array[1..20] of integer;

associates the type expression array(1..20, integer) with C.

- Product: T1 × T2
- If T1 and T2 are two type expressions, then their Cartesian product T1 × T2 is a type expression. We assume that × associates to the left.
- Record: record((N1 × T1) × (N2 × T2))

A record differs from a product. The fields of a record have names. The record type constructor will be applied to a tuple formed from field types and field names. For example, the Pascal program fragment

```
type node = record
address: integer ;
    data : array [1..15] of char
end;
var node table : array [1..10] of node ;
```

declares the type name “node” representing the type expression
    record((address×integer) × (data × array(1..15,char)))
and the variable “node_table” to be an array of records of this type.
Pointer: pointer (T)

Pointer(T) is a type expression denoting the type "pointer to an object of type T where T is a type expression. For example, in Pascal, the declaration

var ptr: *row

declares variable "ptr" to have type pointer(row).

Function: D → R

Mathematically, a function is a mapping from elements of one set called domain to another set called range. We may treat functions in programming languages as mapping a domain type "Dom" to a range type "Rg.". The type of such a function will be denoted by the type expression Dom → Rg. For example, the built-in function mod, i.e. modulus of Pascal has type expression int × int → int. As another example, the Pascal declaration

function fun(a, b: char) * integer;

says that the domain type of function "fun" is denoted by "char × char" and range type by "pointer(integer)." The type expression of function "fun" is thus denoted as follows:

char × char → pointer(integer)

However, there are some languages like Lisp that allow functions to return objects of arbitrary types. For example, we can define a function "g" of type (integer → integer) → (integer → integer).

That is, function "g" takes as input a function that maps an integer to an integer and "g" produces another function of the same type as output.

1.3 Design Of Simple Type Checker

Different type systems are designed for different languages. The type checking can be done in two ways. The checking done at compile time is called static checking and the checking done at run time is called dynamic checking. A system is said to be a Sound System if it completely eliminates the dynamic check. In such systems, if the type checker assigns any type other than type error for some fragment of code, then there is no need to check for errors when it is run. Practically this is not always true; for example, if an array X is declared to hold 100 elements. Usually the index would be from 0 to 99 or from 1 to 100 depending on the language support. And there is a statement in the program referred to as X[i]; during compilation this would not guarantee error free at runtime as there is possibility that if the value of i is 120 at run time then there will be an error. Therefore, it is essential that there is a need even for the dynamic check to be done.

Let us consider a simple language that has declaration statements followed by statements, where these statements are simple arithmetic statements, conditional statements, iterative statements, and functional statements. The program block of code can be generated by defining the rules as follows:

Type Declarations

	P → D ";" E	
	D → D ";" D	
	| id ":" T	{add_type(id.entry, T.type) }
	T → char	{T.type := char }
	T → integer	{T.type := int }
	:…..	:…..
	T → "*" T_1	{T.type := pointer(T_1.type) }
	T → array "["num "]" of T_1	{T.type := array(num.value, T_1.type) }

Table 3.1: Type Declarations

These rules are defined to write the declaration statements followed by expression statements. The semantic rule { add_type(id.entry, T.type) } indicates to associate type in T with the identifier and add this type info into the symbol table during parsing. A semantic rule of the form {T.type := int } associates the type of T to integer. So the above SDT collects type information and stores in symbol table.

1.4 TYPE CHECKING OF EXPRESSIONS

Let us see how to type check expressions. The expressions like 3 mod 5, A[10], *p can be generated by the following rules. The semantic rules are defined as follows to extract the type information and to check for compatibility.

	E → literal	{E.type := char}
	E → num	{E.type := int}
	E → id	{E.type := lookup(id.entry)}
	E → E1 mod E2	{E.type := if E1.type = int and E2.type = int
		then int
		else type_error}
	E → E1 "[" E2 "]"	{E.type := if E1.type = array(s, t) and E2.type = int
		Th en t
		else type_error}
	E → "*" E1	{E.type := if E1.type = pointer(t)
		then t
		else type_error}

Table 3.2: Type Checking Of Expressions

When we write a statement as i mod 10**,** then while parsing the element i, it uses the rule as E → id and performs the action of getting the data type for the id from the symbol table using the lookup method. When it parses the lexeme 10, it uses the rule E → num and assigns the type as int. While parsing the complete statement i mod 10**,** it uses the rule E → E_1 mod E_2, which checks the data types in both E_1 and E_2 and if they are the same it returns int otherwise type_error.

1.5 TYPE CHECKING OF STATEMENTS

The statements are simple of the form "a = b + c" or "a = b." It can be a combination of statements followed by another statement or a conditional statement or iterative. To generate either a simple or a complex group of statements, the rules can be framed as follows: To validate the statement a special data type **void** is defined, which is assigned to a statement only when it is valid at expression level, otherwise type_error is assigned to indicate that it is invalid. If there is an error at expression level, then it is propagated to the statement, from the statement it is propagated to a set of statements and then to the entire block of program.

P → D ";" S	
S → id ":=" E	{S.type := if lookup(id.entry)= E.type
S → S_1 ";" S_2	then void
	else type_error}
	{S.type := if S_1.type = void and S_2.type
	= void
	then void
	else type_error}
S → if E then S_1	{S.type := if E.type = boolean
	then S_1.type
	else type_error}
S → while E do S_1	{S.type := if E.type = boolean
	then S_1.type
	else type_error}

Table 3.3: Type Checking Of Statements

1.6 Type Conversion

In an expression, if there are two operands of different type, then it may be required to convert one type to another in order to perform the operation. For example, the expression "a + b," if a is of integer and b is real, then to perform the addition a may be converted to real. The type checker can be designed to do this conversion. The conversion done automatically by the compiler is implicit conversion and this process is known as coercion. If the compiler insists the programmer to specify this conversion, then it is said to be explicit. For instance, all conversions in Ada are said to be explicit. The semantic rules for type conversion are listed below.

	E → num	{E.type := int}
	E → num.num	{E.type := real}
	E → id	{E.type := lookup(id.entry)}

	E → E1 op E2	{E.type := if E1.type = int and E2.type = int
		then int
		else if E1.type = int and E2.type = real
		then real
		else if E1.type = real and E2.type = int
		then real
		else if E1.type = real and E2.type = real
		then real
		else type_error}

Table 3.4:Type Conversion

2. Overloading Of Functions And Operators

An operator is overloaded if the same operator performs different operations. For example, in arithmetic expression a + b, the addition operator "+" is overloaded because it performs different operations, when a and bare of different types like integer, real, complex, and so on. Another example of operator overloading is overloaded parenthesis in ada, that i, the expression A(i) has different meanings. It can be the i^{th} element of an array, or a call to function A with argument I, and so on. Operator overloading is resolved when the unique definition for an overloaded operator is determined. The process of resolving overloading is called operator identification because it specifies what operation an operator performs. The overloading of arithmetic operators can be easily resolved by processing only the operands of an operator.

Like operator overloading, the function can also be overloaded. In function overloading, the functions have the same name but different numbers and arguments of different types. In Ada, the operator "*" has the standard meaning that it takes a pair of integers and returns an integer. The function of "*" can be overloaded by adding the following declarations:

Function "*"(a,b: integer) return integer.

Function "*"(a,b: complex) return integer.

Function "*"(a,b: complex) return complex.

By addition of the above declarations, now the operator "*" can take the following possible types:

- It takes a pair of integers and returns an integer
- It takes a pair of integers and returns a complex number
- It takes a pair of complex numbers and returns a complex number

Function overloading can be resolved by the type checker based on the number and types of arguments.

The type checking rule for function by assuming that each expression has a unique type is given as

```
E → E1(E2)
{
E.type : = t
E2.type : = t → u then
E.type : = u
else E.type : = type_error
```

}

	$E' \rightarrow E$	{E′.type := E.type}
	$E \rightarrow id$	{E.type := lookup(id.entry)}
	$E \rightarrow E_1(E_2)$	{E.type := { u | there exists an s in E_2.type Such that $s \rightarrow u$ is in E_1.type }

Table 3.5: Overloading Of Functions and Operators

3. Polymorphic Functions

A piece of code is said to be polymorphic if the statements in the body can be executed with different types. A function that takes the arguments of different types and executes the same code is a polymorphic function. The type checker designed for a language like Ada that supports polymorphic functions, the type expressions are extended to include the expressions that vary with type variables. The same operation performed on different types is called overloading and are often found in object-oriented programming. For example, let us consider the function that takes two arguments and returns the result.

int add(int, int)
int add(real, real)
real add(real, real)

The type expression for the function add is given as

$\text{int} \times \text{int} \rightarrow \text{int}$
$\text{real} \times \text{real} \rightarrow \text{int}$
$\text{real} \times \text{real} \rightarrow \text{real}$

Write type expression for an array of pointer to real, where the array index ranges from 1 to 100.

Solution: The type expression is array [1...100, pointer (real)]

Write a type expression for a two-dimensional array of integers (that is, an array of arrays) whose rows are indexed from 0 to 9 and whose columns are indexed from –10 to 10.

Solution: Type expression is array [0..9, array[-10..10,integer]]

Write a type expression for a function whose domains are functions from integers to pointers to integers and whose ranges are records consisting of an integer and a character.

Solution: Type expression is
Domain type expression is integer $\rightarrow$ pointer(integer)
Let range has two fields a and b of type integer and character respectively.
Range type expression is record((a $\times$ integer)(b $\times$ character))
The final type expression is (integer $\rightarrow$ pointer(integer)) $\rightarrow$ record((a $\times$ integer) (b $\times$ character))

Consider the following program in C and the write the type expression for abc.

```
typedef struct
{
    int a,b;
 }   NODE;
    NODE abc[100];
```

Solution: The type expression for NODE is record((a $\times$ integer) $\times$ (b $\times$ integer)) abc is an array of NODE; hence, its type expression is

array[0..99, record((a × integer) × (b × integer))]

Consider the following declarations.

	type cell=record		
		info: integer;	
		next: pointer(cell)	
	Type	link = ↑ cell;	
	Var	next = link;	
		last = link;	
		p = ↑ cell;	
		q,r = ↑ cell;	

Table 3.6: Solution

Among the following, which expressions are structurally equivalent? Which are name equivalent? Justify your answer.

1. link
2. Pointer(cell)
3. Pointer(link)
4. Pointer (record ((info × integer) × (next × pointer (cell))).

Solution: Let	A = link
	B = pointer (cell)
	C = pointer (link)
	D = Pointer (record ((info × integer) × (next ×pointer(cell))).

Table 3.7: Solution

To get structural equivalence we need to substitute each type name by its type expression.
We know that, link is a type name. If we substitute pointer (cell) for each appearance of link we get,
A = pointer (cell)
B = pointer (cell)
C = pointer (pointer (cell))
D = Pointer (record ((info × integer) × (next × pointer (cell))).
We know that, cell is also type name given by
type cell=record
  info: integer;
  next: pointer(cell)
Substituting type expression for cell in A and B, we get

A = pointer (record ((info × integer) × (next × pointer (cell)))
B = pointer (record ((info × integer) × (next × pointer (cell)))
C = pointer(pointer(cell))
D = Pointer (record ((info × integer) × (next × pointer (cell))).
We have not substituted for the type expression of cell in "C" as it is anyway different from A, B, and D. That is, even if we substitute in C, the type expression will not be the same for A, B, C, and D.
We can say that A, B, and D are structurally equivalent.
For name equivalence, we will not do any substitutions. Rather we look at type expressions directly. If they are the same then we say they are name equivalent.
None of A, B, C, D are name equivalent.

4 Run Time Environment

4.1Storage Allocation Information

- Information about storage locations is kept in the symbol table
- If target is assembly code then assembler can take care of storage for various names
- Compiler needs to generate data definitions to be appended to assembly code
- If target is machine code then compiler does the allocation
- For names whose storage is allocated at runtime no storage allocation is done

Information about the storage locations that will be bound to names at run time is kept in the symbol table. If the target is assembly code, the assembler can take care of storage for various names. All the compiler has to do is to scan the symbol table, after generating assembly code, and generate assembly language data definitions to be appended to the assembly language program for each name. If machine code is to be generated by the compiler, then the position of each data object relative to a fixed origin must be ascertained. The compiler has to do the allocation in this case. In the case of names whose storage is allocated on a stack or heap, the compiler does not allocate storage at all, it plans out the activation record for each procedure.

4.2 Storage Organization:

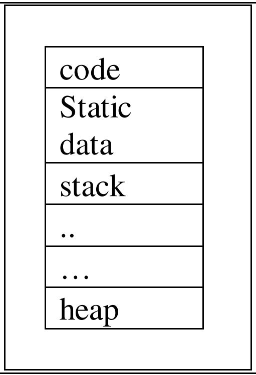

Figure 3.1:Storage stack

This kind of organization of run-time storage is used for languages such as FORTRAN, Pascal and C. The size of the generated target code, as well as that of some of the data objects, is known at compile time. Thus, these can be stored in statically determined areas in the memory. Pascal and C use the stack for procedure activations. Whenever a procedure is called, execution of activation gets interrupted, and information about the machine state (like register values) is stored on the stack. When the called procedure returns, the interrupted activation can be restarted after restoring the saved machine state. The heap may be used to store dynamically allocated data objects, and also other stuff such as activation information (in the case of languages where an activation tree cannot be used to represent lifetimes). Both the stack and the heap change in size during program execution, so they cannot be allocated a fixed amount of space.

Generally they start from opposite ends of the memory and can grow as required, towards each other, until the space available has filled up.

Activation Record:

. **temporaries:** used in expression evaluation
. **local data:** field for local data
. **saved machine status:** holds info about machine status before procedure call
. **access link :** to access non local data
. **control link :** points to activation record of caller
. **actual parameters:** field to hold actual parameters
. **returned value :** field for holding value to be returned

Figure 3.2: Storage Organization

The activation record is used to store the information required by a single procedure call. Not all the fields shown in the figure may be needed for all languages. The record structure can be modified as per the language/compiler requirements. For Pascal and C, the activation record is generally stored on the run-time stack during the period when the procedure is executing. Of the fields shown in the figure, access link and control link are optional (e.g. FORTRAN doesn't need access links). Also, actual parameters and return values are often stored in registers instead of the activation record, for greater efficiency. The activation record for a procedure call is generated by the compiler. Generally, all field sizes can be determined at compile time. However, this is not possible in the case of a procedure which has a local array whose size depends on a parameter. The strategies used for storage allocation in such cases will be discussed in the coming slides.

4.3 Storage Allocation Strategies

These represent the different storage-allocation strategies used in the distinct parts of the run-time memory organization (as shown in slide 8). We will now look at the possibility of using these strategies to allocate memory for activation records. Different languages use different strategies for this purpose. For example, old FORTRAN used static allocation, Algol type languages use stack allocation, and LISP type languages use heap allocation.

Static allocation:

Names are bound to storage as the program is compiled
No runtime support is required
Bindings do not change at run time
On every invocation of procedure names are bound to the same storage
Values of local names are retained across activations of a procedure

These are the fundamental characteristics of static allocation. Since name binding occurs during compilation, there is no need for a run-time support package. The retention of local name values across procedure activations means that when control returns to a procedure, the values of the locals are the same as they were when control last left. For example, suppose we had the following code, written in a language using static allocation: function F( )

```
{
int a;
print(a);
a = 10;
}
```

After calling F( ) once, if it was called a second time, the value of a would initially be 10, and this is what would get printed.

Type of a name determines the amount of storage to be set aside
. Address of a storage consists of an offset from the end of an activation record
. Compiler decides location of each activation
. All the addresses can be filled at compile time
. Constraints
- Size of all data objects must be known at compile time
- Recursive procedures are not allowed
- Data structures cannot be created dynamically
The type of a name determines its storage requirement, as outlined in slide 11. The address for this storage is an offset from the procedure's activation record, and the compiler positions the records relative to the target code and to one another (on some computers, it may be possible to leave this relative position unspecified, and let the link editor link the activation records to the executable code). After this position has been decided, the addresses of the activation records, and hence of the storage for each name in the records, are fixed. Thus, at compile time, the addresses at which the target code can find the data it operates upon can be filled in. The addresses at which information is to be saved when a procedure call takes place are also known at compile time. Static allocation does have some limitations:

- Size of data objects, as well as any constraints on their positions in memory, must be available at compile time.
- No recursion, because all activations of a given procedure use the same bindings for local names.
- No dynamic data structures, since no mechanism is provided for run time storage allocation.

Stack Allocation
The activation records that are pushed onto and popped for the run time stack as the control flows through the given activation tree. First the procedure is activated. Procedure read array's activation is pushed onto the stack, when the control reaches the first line in the procedure sort. After the control returns from the activation of the read array, its activation is popped. In the activation of sort, the control then reaches a call of qsort with actuals 1 and 9 and an activation of qsort is pushed onto the top of the stack. In the last stage the activations for partition (1,3) and qsort (1,0) have begun and ended during the life time of qsort (1,3), so their activation records have come and gone from the stack, leaving the activation record for qsort (1,3) on top.

Calling Sequence:

A call sequence allocates an activation record and enters information into its field
A return sequence restores the state of the machine so that calling procedure can continue execution

Figure 3.3: Calling Sequence in Stack

A call sequence allocates an activation record and enters information into its fields. A return sequence restores the state of the machine so that the calling sequence can continue execution. Calling sequence and activation records differ, even for the same language. The code in the calling sequence is often divided between the calling procedure and the procedure it calls. There is no exact division of runtime tasks between the caller and the callee. As shown in the figure, the register stack top points to the end of the machine status field in the activation record. This position is known to the caller, so it can be made responsible for setting up stack top before control flows to the called procedure. The code for the callee can access its temporaries and the local data using offsets from stack top.
Call Sequence:
Caller evaluates the actual parameters

Caller stores return address and other values (control link) into callee's activation record
Callee saves register values and other status information
Callee initializes its local data and begins execution
The fields whose sizes are fixed early are placed in the middle. The decision of whether or not to use the control and access links is part of the design of the compiler, so these fields can be fixed at compiler construction time. If exactly the same amount of machine-status information is saved for each activation, then the same code can do the saving and restoring for all activations. The size of temporaries may not be known to the front end. Temporaries needed by the procedure may be reduced by careful code generation or optimization. This field is shown after that for the local data. The caller usually evaluates the parameters and communicates them to the activation record of the callee. In the runtime stack, the activation record of the caller is just below that for the callee. The fields for parameters and a potential return value are placed next to the activation record of the caller. The caller can then access these fields using offsets from the end of its own activation record. In particular, there is no reason for the caller to know about the local data or temporaries of the callee.

Return Sequence:
Callee places a return value next to activation record of caller
Restores registers using information in status field
Branch to return address
Caller copies return value into its own activation record
As described earlier, in the runtime stack, the activation record of the caller is just below that for the callee. The fields for parameters and a potential return value are placed next to the activation record of the caller. The caller can then access these fields using offsets from the end of its own activation record. The caller copies the return value into its own activation record. In particular, there is no reason for the caller to know about the local data or temporaries of the callee. The given calling sequence allows the number of arguments of the called procedure to depend on the call. At compile time, the target code of the caller knows the number of arguments it is supplying to the callee. The caller knows the size of the parameter field. The target code of the called must be prepared to handle other calls as well, so it waits until it is called, then examines the parameter field. Information describing the parameters must be placed next to the status field so the callee can find it.

Long Length Data
The procedure P has three local arrays. The storage for these arrays is not part of the activation record for P; only a pointer to the beginning of each array appears in the activation record. The relative addresses of these pointers are known at the compile time, so the target code can access array elements through the pointers. Also shown is the procedure Q called by P. The activation record for Q begins after the arrays of P. Access to data on the stack is through two pointers, top and stack top. The first of these marks the actual top of the stack; it points to the position at which the next activation record begins. The second is used to find the local data. For consistency with the organization of the figure in slide 16, suppose the stack top points to the end of the machine status field. In this figure the stack top points to the end of this field in the activation record for Q. Within the field is a control link to the previous value of stack top when control was in calling activation of P. The code that repositions top and stack top can be generated at compile time, using the sizes of the fields in the activation record. When q returns, the new value of top is stack top minus the length of the machine status and the parameter fields in Q's activation record. This length is known at the compile time, at least to the caller. After adjusting top, the new value of stack top can be copied from the control link of Q.

Dangling references:
Referring to locations which have been deallocated

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

The problem of dangling references arises, whenever storage is de-allocated. A dangling reference occurs when there is a reference to storage that has been de-allocated. It is a logical error to use dangling references, since the value of de-allocated storage is undefined according to the semantics of most languages. Since that storage may later be allocated to another datum, mysterious bugs can appear in the programs with dangling references.

Heap Allocation:

Stack allocation cannot be used if:
The values of the local variables must be retained when an activation ends
A called activation outlives the caller
In such a case de-allocation of activation record cannot occur in last-in first-out fashion
Heap allocation gives out pieces of contiguous storage for activation records
There are two aspects of dynamic allocation -:
Runtime allocation and de-allocation of data structures.
Languages like Algol have dynamic data structures and it reserves some part of memory for it.
If a procedure wants to put a value that is to be used after its activation is over then we cannot use stack for that purpose. That is language like Pascal allows data to be allocated under program control. Also in certain language a called activation may outlive the caller procedure. In such a case last-in-first-out queue will not work and we will require a data structure like heap to store the activation. The last case is not true for those languages whose activation trees correctly depict the flow of control between procedures.
Pieces may be de-allocated in any order
Over time the heap will consist of alternate areas that are free and in use
Heap manager is supposed to make use of the free space
For efficiency reasons it may be helpful to handle small activations as a special case
For each size of interest keep a linked list of free blocks of that size
Initializing data-structures may require allocating memory but where to allocate this memory. After doing type inference we have to do storage allocation. It will allocate some chunk of bytes. But in language like lisp it will try to give continuous chunk. The allocation in continuous bytes may lead to problem of fragmentation i.e. you may develop hole in process of allocation and de-allocation. Thus storage allocation of heap may lead us with many holes and fragmented memory which will make it hard to allocate continuous chunk of memory to requesting program. So we have heap mangers which manage the free space and allocation and de-allocation of memory. It would be efficient to handle small activations and activations of predictable size as a special case as described in the next slide. The various allocation and de-allocation techniques used will be discussed later.
Fill a request of size s with block of size s ' where s ' is the smallest size greater than or equal to s

- For large blocks of storage use heap manager
- For large amount of storage computation may take some time to use up memory so that time taken by the manager may be negligible compared to the computation time

As mentioned earlier, for efficiency reasons we can handle small activations and activations of predictable size as a special case as follows:
For each size of interest, keep a linked list if free blocks of that size
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 de-allocated, it is returned to the linked list it came from.
For large blocks of storage use the heap manger.

Heap manger will dynamically allocate memory. This will come with a runtime overhead. As heap manager will have to take care of defragmentation and garbage collection. But since heap manger saves space otherwise we will have to fix size of activation at compile time, runtime overhead is the price worth it.

Access to non-local names :
Scope rules determine the treatment of non-local names
A common rule is lexical scoping or static scoping (most languages use lexical scoping)
The scope rules of a language decide how to reference the non-local variables. There are two methods that are commonly used:

1. Static or Lexical scoping: It determines the declaration that applies to a name by examining the program text alone. E.g., Pascal, C and ADA.
2. Dynamic Scoping: It determines the declaration applicable to a name at run time, by considering the current activations. E.g., Lisp

5. Dynamic Storage Allocation:
Generally languages like Lisp and ML which do not allow for explicit de-allocation of memory do garbage collection. A reference to a pointer that is no longer valid is called a 'dangling reference'. For example, consider this C code:

```
int main (void)
{
int* a=fun();
}
int* fun()
{
int a=3;
int* b=&a;
return b;
}
```

Here, the pointer returned by fun() no longer points to a valid address in memory as the activation of fun() has ended. This kind of situation is called a 'dangling reference'. In case of explicit allocation it is more likely to happen as the user can de-allocate any part of memory, even something that has to a pointer pointing to a valid piece of memory.
Explicit Allocation of Fixed Sized Blocks
Link the blocks in a list
Allocation and de-allocation can be done with very little overhead
The simplest form of dynamic allocation involves blocks of a fixed size. By linking the blocks in a list, as shown in the figure, allocation and de-allocation can be done quickly with little or no storage overhead.
Explicit Allocation of Fixed Sized Blocks
Blocks are drawn from contiguous area of storage
An area of each block is used as pointer to the next block
A pointer available points to the first block
Allocation means removing a block from the available list
De-allocation means putting the block in the available list
Compiler routines need not know the type of objects to be held in the blocks
Each block is treated as a variant record
Suppose that blocks are to be drawn from a contiguous area of storage. Initialization of the area is done by using a portion of each block for a link to the next block. A pointer availablepoints to the first block. Generally a list of free nodes and a list of allocated nodes is maintained, and whenever a new block has to be allocated, the block at the head of the free list is taken off and allocated (added to the list of allocated

nodes). When a node has to be de-allocated, it is removed from the list of allocated nodes by changing the pointer to it in the list to point to the block previously pointed to by it, and then the removed block is added to the head of the list of free blocks. The compiler routines that manage blocks do not need to know the type of object that will be held in the block by the user program. These blocks can contain any type of data (i.e., they are used as generic memory locations by the compiler). We can treat each block as a variant record, with the compiler routines viewing the block as consisting of some other type. Thus, there is no space overhead because the user program can use the entire block for its own purposes. When the block is returned, then the compiler routines use some of the space from the block itself to link it into the list of available blocks, as shown in the figure in the last slide.
Explicit Allocation of Variable Size Blocks
Storage can become fragmented
Situation may arise
If program allocates five blocksthen de-allocates second and fourth block

Fragmentation is of no consequence if blocks are of fixed size
Blocks cannot be allocated even if space is available
In explicit allocation of fixed size blocks, internal fragmentation can occur, that is, the heap may consist of alternate blocks that are free and in use, as shown in the figure. The situation shown can occur if a program allocates five blocks and then de-allocates the second and the fourth, for example. Fragmentation is of no consequence if blocks are of fixed size, but if they are of variable size, a situation like this is a problem, because we could not allocate a block larger than any one of the free blocks, even though the space is available in principle. So, if variable- sized blocks are allocated, then internal fragmentation can be avoided, as we only allocate as much space as we need in a block. But this creates the problem of external fragmentation, where enough space is available in total for our requirements, but not enough space is available in continuous memory locations, as needed for a block of allocated memory. For example, consider another case where we need to allocate 400 bytes of data for the next request, and the available continuous regions of memory that we have are of sizes 300, 200 and 100 bytes. So we have a total of 600 bytes, which is more than what we need. But still we are unable to allocate the memory as we do not have enough contiguous storage. The amount of external fragmentation while allocating variable-sized blocks can become very high on using certain strategies for memory allocation. So we try to use certain strategies for memory allocation, so that we can minimize memory wastage due to external fragmentation. These strategies are discussed in the next few slides.

6 Symbol Table

Compiler uses symbol table to keep track of scope and binding information about names
Symbol table is changed every time a name is encountered in the source; changes to table occur

- if a new name is discovered
- if new information about an existing name is discovered

Symbol table must have mechanism to:

- add new entries
- find existing information efficiently

Two common mechanisms:

- linear lists, simple to implement, poor performance
- hash tables, greater programming/space overhead, good performance

Compiler should be able to grow symbol table dynamically
if size is fixed, it must be large enough for the largest program

A compiler uses a symbol table to keep track of scope and binding information about names. It is filled after the AST is made by walking through the tree, discovering and assimilating information about the names. There should be two basic operations - to insert a new name or information into the symbol table as and when discovered and to efficiently lookup a name in the symbol table to retrieve its information.

Variable	Information(type)	Space (byte)
A	Integer	2
B	float	4
C	Float	8
D	Character	1
..	..	…..

Table 3.8: Symbol Table

Two common data structures used for the symbol table are -

- Linear lists:- simple to implement, poor performance.
- Hash tables:- greater programming/space overhead, good performance.

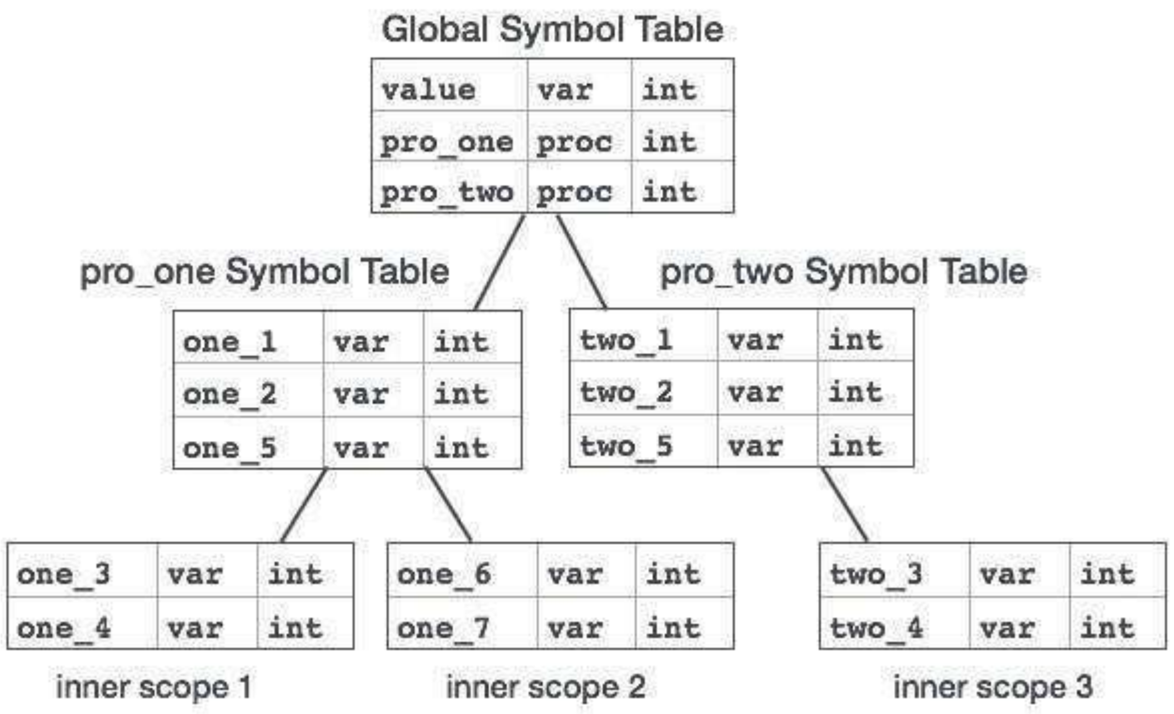

Figure 3.4: Symbol table

Ideally a compiler should be able to grow the symbol table dynamically, i.e., insert new entries or information as and when needed. But if the size of the table is fixed in advance then (an array implementation for example), then the size must be big enough in advance to accommodate the largest possible program.

- each entry for a declaration of a name
- format need not be uniform because information depends upon the usage of the name
- each entry is a record consisting of consecutive words
- to keep records uniform some entries may be outside the symbol table
- information is entered into symbol table at various times
- keywords are entered initially
- identifier lexemes are entered by lexical analyzer
- symbol table entry may be set up when role of name becomes clear
- attribute values are filled in as information is available

For each declaration of a name, there is an entry in the symbol table. Different entries need to store different information because of the different contexts in which a name can occur. An entry corresponding to a particular name can be inserted into the symbol table at different stages depending on when the role of the name becomes clear. The various attributes that an entry in the symbol table can have are lexeme, type of name, size of storage and in case of functions - the parameter list etc.
a name may denote several objects in the same block
int x; struct x {float y, z; }

- lexical analyzer returns the name itself and not pointer to symbol table entry
- record in the symbol table is created when role of the name becomes clear
- in this case two symbol table entries will be created
- attributes of a name are entered in response to declarations
- labels are often identified by colon
- syntax of procedure/function specifies that certain identifiers are formals
- characters in a name
- there is a distinction between token id, lexeme and attributes of the names
- it is difficult to work with lexemes
- if there is modest upper bound on length then lexemes can be stored in symbol table
- if limit is large store lexemes separately

There might be multiple entries in the symbol table for the same name, all of them having different roles. It is quite intuitive that the symbol table entries have to be made only when the role of a particular name becomes clear. The lexical analyzer therefore just returns the name and not the symbol table entry as it cannot determine the context of that name. Attributes corresponding to the symbol table are entered for a name in response to the corresponding declaration. There has to be an upper limit for the length of the lexemes for them to be stored in the symbol table.