PUNE VIDYARTHI GRIHA's

COLLEGE OF ENGINEERING

(Approved by AICTE, Accredited by NAAC, Affiliated to SPPU)

NASHIK – 04.

COMPILER NOTES

UNIT - IV

DEPARTMENT OF COMPUTER ENGINEERING

AY – 2018-19

UNIT – IV

Syllabus - Storage Management – Static, Stack and Heap, Activation Record, static and control links, parameter passing, return value, passing array and variable number of arguments, Static and Dynamic scope, Dangling Pointers, translation of control structures – if, if-else statement, Switch-case, while, do -while statements, for, nested blocks, display mechanism, array assignment, pointers, function call and return. Translation of OO constructs: Class, members and Methods.

RUN-TIME STORAGE MANAGEMENT

6.1 RUNTIME ENVIRONMENT

- Runtime organization of different storage locations
- Representation of scopes and extents during program execution.
- Components of executing program reside in blocks of memory (supplied by OS).
- Three kinds of entities that need to be managed at runtime:

  - Generated code for various procedures and programs.

- forms text or code segment of your program: size known at compile time.

  - Data objects:

- Global variables/constants: size known at compile time •
Variables declared within procedures/blocks: size known •
Variables created dynamically: size unknown.

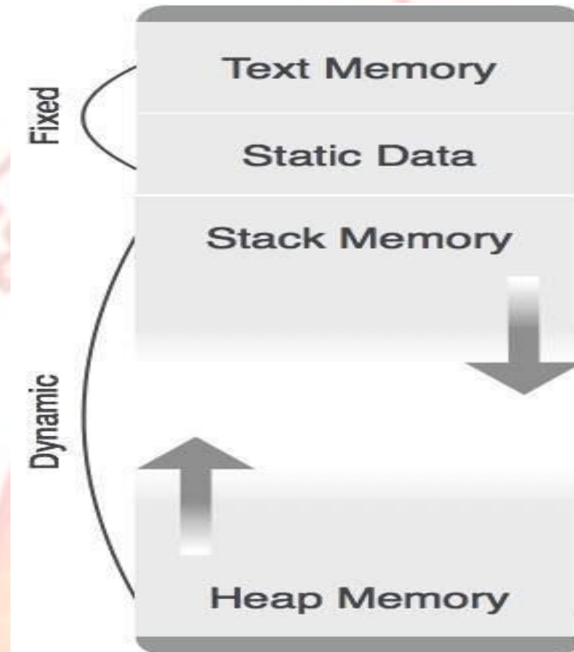
  - Stack to keep track of procedure activations. Subdivide

- memory conceptually into code and data areas:

    - Code: Program •
instructions
    - Stack: Manage activation of procedures at runtime.
    - Heap: holds variables created dynamically

6.2 STORAGE ORGANIZATION

1*Fixed-size objects can be placed in predefined locations.*



2. Run-time stack and heap The
   STACK is used to store:
   - Procedure activations.
   - The status of the machine just before calling a procedure, so that the status can be restored
   - when the called procedure returns.
   - The HEAP stores data allocated under program control (e.g. by malloc() in C).

I. Static Storage Allocation

- For any program if we create memory at compile time, memory will be created in the static area.
- For any program if we create memory at compile time only, memory is created only once.
- It don't support dynamic data structure i.e memory is created at compile time and deallocated after program completion.
- The drawback with static storage allocation is recursion is not supported.
- Another drawback is size of data should be known at compile time

Eg- FORTRAN was designed to permit static storage allocation.

II. Stack Storage Allocation

- Storage is organised as a stack and activation records are pushed and popped as activation begin and end respectively. Locals are contained in activation records so they are bound to fresh storage in each activation.
- Recursion is supported in stack allocation

III. **Heap Storage Allocation**

- Memory allocation and deallocation can be done at any time and at any place depending on the requirement of the user.
- Heap allocation is used to dynamically allocate memory to the variables and claim it back when the variables are no more required.
- Recursion is supported.

Activation Record/Tree :

A program is a sequence of instructions combined into a number of procedures. Instructions in a procedure are executed sequentially. A procedure has a start and an end delimiter and everything inside it is called the body of the procedure. The procedure identifier and the sequence of finite instructions inside it make up the body of the procedure.

The execution of a procedure is called its activation. An activation record contains all the necessary information required to call a procedure. An activation record may contain the following units (depending upon the source language used).

Temporaries	Stores temporary and intermediate values of an expression.
Local Data	Stores local data of the called procedure.
Machine Status	Stores machine status such as Registers, Program Counter etc., before the procedure is called.
Control Link	Stores the address of activation record of the caller procedure.
Access Link	Stores the information of data which is outside the local scope.
Actual Parameters	Stores actual parameters, i.e., parameters which are used to send input to the called procedure.
Return Value	Stores return values.

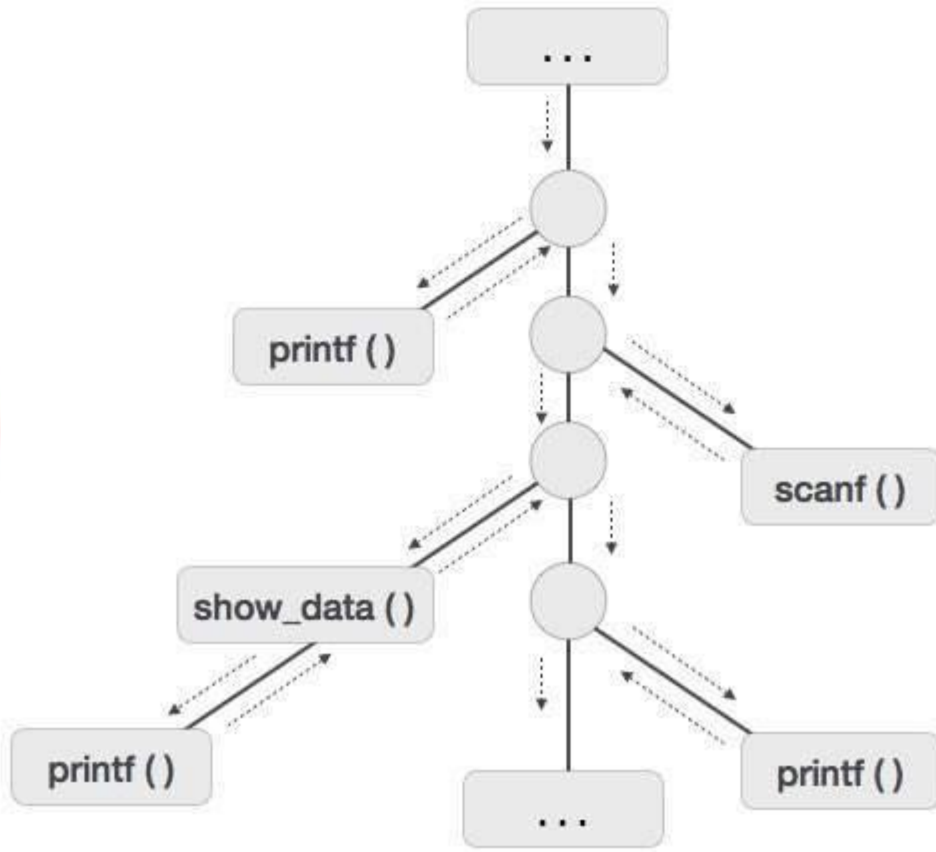
Whenever a procedure is executed, its activation record is stored on the stack, also known as control stack. When a procedure calls another procedure, the execution of the caller is suspended until the called procedure finishes execution. At this time, the activation record of the called procedure is stored on the stack.

We assume that the program control flows in a sequential manner and when a procedure is called, its control is transferred to the called procedure. When a called procedure is executed, it returns the control back to the caller. This type of control flow makes it easier to represent a series of activations in the form of a tree, known as the **activation tree**.

To understand this concept, we take a piece of code as an example:

```
. . .
printf("Enter Your Name: ");
scanf("%s", username);
show_data(username);
printf("Press any key to continue…");
. . .
int show_data(char *user)
  {
  printf("Your name is %s", username);
  return 0;
  }
. . .
```

Below is the activation tree of the code given.



Now we understand that procedures are executed in depth-first manner, thus stack allocation is the best suitable form of storage for procedure activations.

6.3 PARAMETERS PASSING

A language has first-class functionsif functions can bedeclared within any scope passed as arguments to other functions returned as results of functions.�In a language with first-class

functions and static scope, a function value is generally represented by a closure. a pair consisting of a pointer to function code a pointer to an activation record.�Passing functions as arguments is very useful in structuring of systems using upcalls

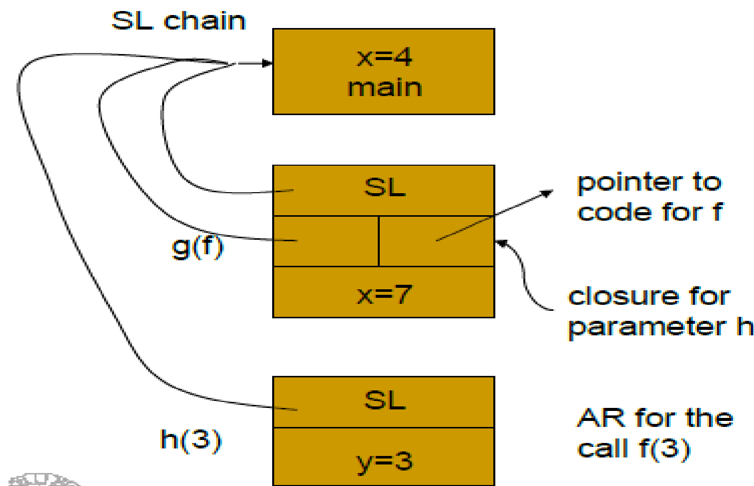
An example:

```
main()
{ int x = 4; int f
(int y) { return
x*y;
}
int g (int →int h){ int x
= 7;
return h(3) + x;
}
```

g(f);//returns 12

}

Passing Functions as Parameters – Implementation with Static Scope




```
An example:
main()
{ int x = 4;
  int f (int y) {
    return x*y;
  }
  int g (int → int h){
    int x = 7;
    return h(3) + x;
  }
  g(f);// returns 12
}
```

Parameter Passing

The communication medium among procedures is known as parameter passing. The values of the variables from a calling procedure are transferred to the called procedure by some mechanism. Before moving ahead, first go through some basic terminologies pertaining to the values in a program.

r-value

The value of an expression is called its r-value. The value contained in a single variable also becomes an r-value if it appears on the right-hand side of the assignment operator. r-values can always be assigned to some other variable.

l-value

The location of memory (address) where an expression is stored is known as the l-value of that expression. It always appears at the left hand side of an assignment operator.

For example:

```
day = 1;
```

```
week = day * 7;

month = 1;

year = month * 12;
```

From this example, we understand that constant values like 1, 7, 12, and variables like day, week, month and year, all have r-values. Only variables have l-values as they also represent the memory location assigned to them.

For example:

```
7 = x + y;
```

is an l-value error, as the constant 7 does not represent any memory location.

Formal Parameters

Variables that take the information passed by the caller procedure are called formal parameters. These variables are declared in the definition of the called function.

Actual Parameters

Variables whose values or addresses are being passed to the called procedure are called actual parameters. These variables are specified in the function call as arguments.

Example:

```
fun_one()

{

  int actual_parameter = 10;

  call fun_two(int actual_parameter);

}

  fun_two(int formal_parameter)

{

  print formal_parameter;

}
```

Formal parameters hold the information of the actual parameter, depending upon the parameter passing technique used. It may be a value or an address.

Pass by Value

In pass by value mechanism, the calling procedure passes the r-value of actual parameters and the compiler puts that into the called procedure's activation record. Formal parameters then hold the values passed by the calling procedure. If the values held by the formal parameters are changed, it should have no impact on the actual parameters.

Pass by Reference

In pass by reference mechanism, the l-value of the actual parameter is copied to the activation record of the called procedure. This way, the called procedure now has the address (memory location) of the actual parameter and the formal parameter refers to the same memory location. Therefore, if the value pointed by the formal parameter is changed, the impact should be seen on the actual parameter as they should also point to the same value.

Pass by Copy-restore

This parameter passing mechanism works similar to 'pass-by-reference' except that the changes to actual parameters are made when the called procedure ends. Upon function call, the values of actual parameters are copied in the activation record of the called procedure. Formal parameters if manipulated have no real-time effect on actual parameters (as l-values are passed), but when the called procedure ends, the l-values of formal parameters are copied to the l-values of actual parameters.

Example:

```
int y;
calling_procedure()
{
  y = 10;
  copy_restore(y); //l-value of y is passed
  printf y; //prints 99
}
copy_restore(int x)
{
  x = 99; // y still has value 10 (unaffected)
  y = 0; // y is now 0
}
```

When this function ends, the l-value of formal parameter x is copied to the actual parameter y. Even if the value of y is changed before the procedure ends, the l-value of x is copied to the l-value of y making it behave like call by reference.

Pass by Name

Languages like Algol provide a new kind of parameter passing mechanism that works like preprocessor in C language. In pass by name mechanism, the name of the procedure being called is replaced by its actual body. Pass-by-name textually substitutes the argument expressions in a procedure call for the corresponding parameters in the body of the procedure so that it can now work on actual parameters, much like pass-by-reference.

6.4 Static and Dynamic Scope :

The **scope** of a variable x is the region of the program in which uses of x refers to its declaration. One of the basic reasons of scoping is to keep variables in different parts of program distinct from one another. Since there are only a small number of short variable names, and programmers share habits about naming of variables (e.g., i for an array index), in any program of moderate size the same variable name will be used in multiple different scopes.

Scoping is generally divided into two classes:

1.Static Scoping

2.Dynamic Scoping

Static Scoping:

Static scoping is also called **lexical scoping**. In this scoping a variable always refers to its top level environment. This is a property of the program text and unrelated to the run time call stack. Static scoping also makes it much easier to make a modular code as programmer can figure out the scope just by looking at the code. In contrast, dynamic scope requires the programmer to anticipate all possible dynamic contexts.

In most of the programming languages including C, C++ and Java, variables are always statically (or lexically) scoped i.e., binding of a variable can be determined by program text and is independent of the run-time function call stack.

For example, output for the below program is 10, i.e., the value returned by f() is not dependent on who is calling it (Like g() calls it and has a x with value 20). f() always returns the value of global variable x.

filter_none

edit

play_arrow

brightness_4

// A C program to demonstrate static scoping.

```
#include<stdio.h>
int x = 10;

// Called by g()
int f()
{
  return x;
}

// g() has its own variable
// named as x and calls f()
int g()
{
  int x = 20;
  return f();
}

int main()
{
 printf("%d", g());
 printf("\n");
 return 0;
}
```

Output :

```
10
```

To sum up in static scoping the compiler first searches in the current block, then in the surrounding blocks successively and finally in the global variables.

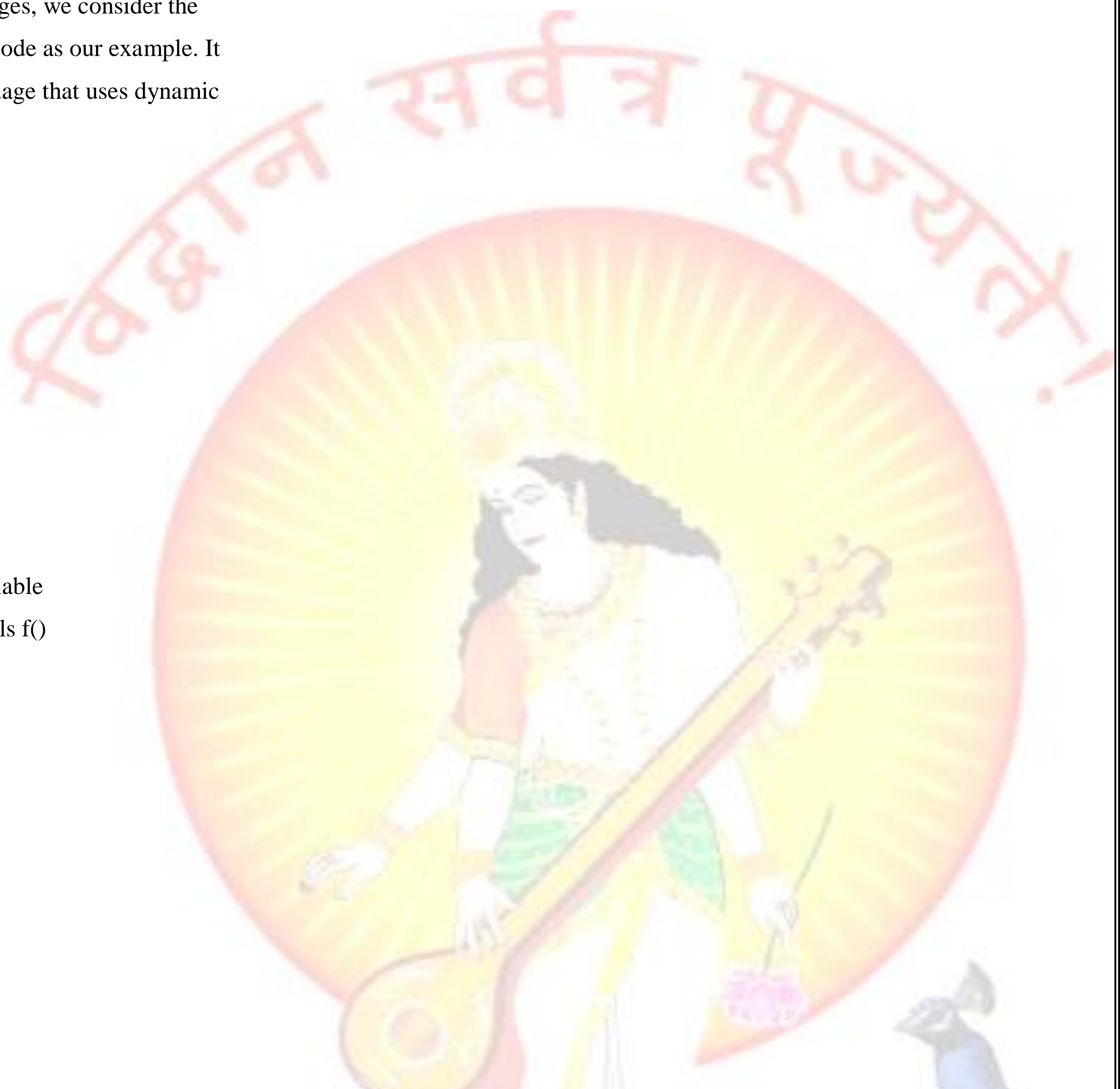
Dynamic Scoping:

With dynamic scope, a global identifier refers to the identifier associated with the most recent environment, and is uncommon in modern languages. In technical terms, this means that each identifier has a global stack of bindings and the occurrence of a identifier is searched in the most recent binding.

In simpler terms, in dynamic scoping the compiler first searches the current block and then successively all the calling functions.

filter_none

brightness_4



```
// Since dynamic scoping is very uncommon in
// the familiar languages, we consider the
// following pseudo code as our example. It
// prints 20 in a language that uses dynamic
// scoping.

int x = 10;

// Called by g()
int f()
{
  return x;
}

// g() has its own variable
// named as x and calls f()
int g()
{
  int x = 20;
  return f();
}

main()
{
 printf(g());
}
```

Output in a a language that uses Dynamic Scoping :

```
20
```

Static Vs Dynamic Scoping

In most of the programming languages static scoping is dominant. This is simply because in static scoping it's easy to reason about and understand just by looking at code. We can see what variables are in the scope just by looking at the text in the editor.

Dynamic scoping does not care how the code is written, but instead how it executes. Each time a new function is executed, a new scope is pushed onto the stack.

Perl supports both dynamic ans static scoping. Perl's keyword "my" defines a statically scoped local variable, while the keyword "local" defines dynamically scoped local variable.

filter_none

brightness_4

```
# A perl code to demonstrate dynamic scoping
$x = 10;
sub f
{
  return $x;
}
sub g
{
  # Since local is used, x uses
  # dynamic scoping.
  local $x = 20;

  return f();
}
print g()."\n";
```

Output :

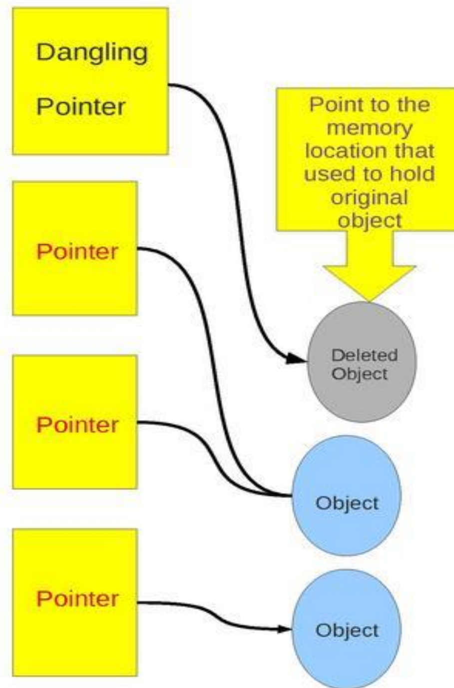
```
20
```

6.5 Dangling Pointer :

Dangling pointers and **wild pointers** in computer programming are pointers that do not point to a valid object of the appropriate type. These are special cases of memory safety violations. More generally, **dangling references** and **wild references** are references that do not resolve to a valid destination, and include such phenomena as link rot on the internet.

Dangling pointers arise during object destruction, when an object that has an incoming reference is deleted or deallocated, without modifying the value of the pointer, so that the pointer still points to the memory location of the deallocated memory. The system may reallocate the previously freed memory, and if the program then dereferences the (now) dangling pointer, *unpredictable behavior may result*, as the memory may now contain completely different data. If the program writes to memory referenced by a dangling pointer, a silent corruption of unrelated data may result, leading to subtle bugs that can be extremely difficult to find. If the memory has been reallocated to another process, then attempting to dereference the dangling pointer can cause segmentation faults (UNIX, Linux) or general protection faults (Windows). If the program has sufficient privileges to allow it to overwrite the bookkeeping data used by the kernel's memory allocator, the corruption can cause system instabilities. In object-oriented languages with garbage collection, dangling references are prevented by only destroying objects that are unreachable, meaning they do not have any incoming pointers;

this is ensured either by tracing or reference counting. However, a finalizer may create new references to an object, requiring object resurrection to prevent a dangling reference



6.6 Flow of control statement :

The main concern with flow control is the additional branching instructions that must fit between the other blocks of code that is represented by simple nonterminals.

stmt → IF expr THEN stmt
| IF expr THEN stmt ELSE stmt
| WHILE expr DO stmt

Selection Statements

Reviewing the pure translation of these statements:

stmt → IF expr THEN stmt1
 	
IF a+b<=c-d THEN
BEGIN
a:=a+1;
writeln(a)
END;
b:=b+2;
	code for expr
	
	if (! mem[???]) goto t1;
	code for stmt1
	
	t1:
	code that follows

stmt → IF expr THEN stmt1 ELSE stmt2
 	
IF a+b<=c-d THEN
BEGIN
a:=a+1;
writeln(a)
END
ELSE BEGIN
d:=d+1;
writeln(d)
END;
b:=b+2;
	code for expr
	
	if (! mem[???]) goto t1;
	code for stmt1
	
	goto t2
	t1:
	code for stmt2
	
	t2:
	code that follows

Timing problem

There is no problem if we generate code for a target language that accepts symbolic labels.

So one solution is to have the output of the compiler be in assembler which then is assembled in a final step.

What we need to do is recognize that some critical code must be generated on encountering the keywords THEN or ELSE. Make up trivial productions to force the parser to perform a reduction on something that would have originally been a shift.

then → THEN

```
{ label = newlabel( )
  cout << "if ( ! mem[ " << ???? << "]) goto "<< label<< endl;
  $$ = label;
}
```

stmt → IF expr then stmt

```
{ cout << $3 << ":") }
```

As exercises determine the semantic actions for handling the else part:

else → ELSE

stmt → IF expr then stmt$_1$ else stmt$_2$

Backpatching

To directly generate machine code, targets of branches must either be known or filled in later, that is, labels are not used to mark the position of an instruction but we must refer to the address of the instruction directly.

Implementation

We need or modify the following abstract functions to manage lists of instructions that need to have their target reference completed later

- *emit (intermediate code)*
  This procedure places each instruction in an array (rather than out to a file) in the same way the program will be in memory within the interpreter. All branches refer to the index of the target instruction.
- *makelist (i)*
  This function creates a new list containing i, initialized with an index into the array of instructions. This function returns a pointer to the list
- *merge(p1,p2)*
  This function concatenates two lists and returns a pointer to the newly merged list.
- *backpatch(p,i)*
  This function inserts i as the target label for each of the instructions in the list pointed to by *p* and removes the list.
- *nextquad*
  This variable holds the address of the next instruction to be emitted.

Flow-of-control statements with backpatching

stmt → IF expr then stmt$_1$

{//full reduction comes later}

then → THEN

```
{ $$.nextlist = makelist(nextaddress); //track backpatch
  emit (JIF,0,0); //jump if false, but to where is unknown
}
```

stmt → IF expr then stmt$_1$

```
{ backpatch($3.nextlist, nextaddress);  //now we know target of JIF
}
```

else → ELSE

```
{ $$.nextlist = makelist (nextaddress);//track backpatch
  emit (JP,0,0)   //branch after true task to endif
  $$.falsetarget = nextaddress; //track backpatch
}
```

stmt → IF expr then stmt$_1$ else stmt$_2$

```
{ backpatch($3.nextlist, $5.falsetarget)  //fix branch to else part
  backpatch($5.nextlist, nextquad)        //fix branch to endif
}
```

stmt → WHILE expr DO stmt$_1$

6.7 Function Calling and Return :

Calling Sequence

The following actions occur during a call.

1. The caller begins the process of creating the callee's AR by evaluating the arguments and placing them in the AR of the callee. (I use arguments for the caller, parameters for the callee.)
2. The caller stores the return address and the (soon-to-be-updated) sp in the callee's AR.
3. The caller increments sp so that instead of pointing into its AR, it points to the corresponding point in the callee's AR.
4. The callee saves the registers and other (system dependent) information.
5. The callee allocates and initializes its local data.

6. The callee begins execution.

Return Sequence

When the procedure returns, the following actions are performed by the callee, essentially undoing the effects of the calling sequence.

1. The callee stores the return value. Note that this address can be determined by the caller using the old (soon-to-be-restored) sp.
2. The callee restores sp and the registers.
3. The callee jumps to the return address.

Note that varagrs are supported.

Also note that the values written during the calling sequence are not erased and the space is not *explicitly* reclaimed. Instead, the sp is restored and, if and when the caller makes another call, the space will be reused.

6.8 Translation of OO : Class , Member and Methods :

Object-oriented methodology is a way of viewing software components and their relationships. Object-oriented methodology relies on three characteristics that define object-oriented languages: encapsulation, polymorphism, and inheritance. These three terms are elaborated below.

Objects and Methods

An *object* is an encapsulation of data together with procedures that manipulate the data and functions that return information about the data. The procedures and functions are both called *methods*.

Encapsulation

Encapsulation refers to mechanisms that allow each object to have its own data and methods. The idea of encapsulating data together with methods existed before object-oriented languages were developed. It is, for example, inherent in the concept of an abstract data type. Objects can be implemented in classical languages such as C using separate compilation or structs to provide the encapsulation. This is a common technique for implementing abstract data types. See Implementing Objects in C for a brief explanation and example.

Object-oriented languages provide more powerful and flexible encapsulation mechanisms for restricting interactions between components. When used carefully, these mechanisms allow the software developers to restrict the interactions between components to those that are required to achieve the desired functionality. The management of component interactions is an important part of software design. It has a significant impact on the ease of understanding, testing, and maintenance of components.

Messages and Receivers

In object-oriented languages, this encapsulation is effected in part by having all method calls handled by objects that recognize the method. This leads to a different syntax for calling methods. For example, in C, you might have a table data structure with a procedure called add for adding a new entry. The declaration for this procedure could be

```
void add(Table t, Data dat, Key k);
```

A typical call for this function is

```
add(t, dat, k);
```

In the object-oriented language Java, a method with similar effect could be declared by

```
void add(Data dat, Key k);
```

and a typical call would be

```
t.add(dat, k);
```

In this call, t plays a special role: it recognizes and carrys out the add method. In object-oriented terminology, this method call is refered to as a *message*, and t is the *receiver* of the message. One benefit of this approach is that there can be many methods named "add", with different objects implementing them in different ways. This allows programmers to reuse names of methods, allowing the same name to have different meanings in different contexts.

Polymorphism and Overloading

Polymorphism refers to the capability of having methods with the same names and parameter types exhibit different behavior depending on the receiver. In other words, you can send the same message to two different objects and they can respond in different ways.

More generally, the capability of using names to mean different things in different contexts is called *overloading*. This also includes allowing two methods to have the same name but different parameters types, with different behavior depending on the parameter types. Note that a language could support some kinds of overloading without supporting polymorphism. In that case, most people in the object-oriented community would not consider it to be an object-oriented language.

Polymorphism and overloading can lead to confusion if used excessively. However, the capability of using words or names to mean different things in different contexts is an important part of the power of natural languages. People begin developing the skills for using it in early childhood.

Members

Objects can have their own data, including variables and constants, and their own methods. The variables, constants, and methods associated with an object are collectively refered to as its *members* or *features*.

Classes

Many object-oriented languages use an important construction called a class. A *class* is a category of objects, classified according to the members that they have. Like objects, classes can also be implemented in classical languages, using separate compilation and structs for encapsulation. See Implementing a Class in C for a brief explanation and example.

The object-oriented language Java uses the following syntax for class definitions:

public class A {

declarations for members

}

Each object in the class will have all members defined in the declarations.

Class Members and Instance Members

In many object-oriented languages, classes are objects in their own right (to a greater or lesser extent, depending on the language). Their primary function is as factories for objects in the category. A class can also hold data variable and constants that are shared by all of its objects and can handle methods that deal with an entire class rather than an individual object. These members are called *class members* or, in some languages (C++ and Java, for example), *static members*. The members that are associated with objects are called *instance members*.

Inheritance

One important characteristic of object-oriented languages is inheritance. *Inheritance* refers to the capability of defining a new class of objects that inherits from a parent class. New data elements and methods can be added to the new class, but the data elements and methods of the parent class are available for objects in the new class without rewriting their declarations.

For example, Java uses the following syntax for inheritance:

```
public class B extends A {

    declarations for new members

}
```

Objects in class B will have all members that are defined for objects in class A. In addition, they have the new members defined in the declaration of class B. The extends keyword signals that class B inherits from class A. We also say that B is a *subclass* of A and that A is the *parent class* of B.

In some languages, Java for example, the programmer has some control over which members are inherited. In Java, a member is defined with a keyword indicating its level of accessibility. The keyword private indicates that the member is *not* inherited by subclasses. This capability is not often used.

TEXT BOOKS:

1. Compilers, Principles Techniques and Tools- Alfred V Aho, Monical S Lam, Ravi Sethi, Jeffrey D. Ullman,2nd ed, Pearson,2007.
2. Principles of compiler design, V. Raghavan, 2nd ed, TMH, 2011.
3. Principles of compiler design, 2nd ed, Nandini Prasad, Elsevier

REFERENCE BOOKS:

1. http://www.nptel.iitm.ac.in/downloads/106108052/
2. Compiler construction, Principles and Practice, Kenneth C Louden, CENGAGE
3. Implementations of Compiler, A new approach to Compilers including the algebraic methods, Yunlinsu, SPRINGER