MODULE 4 – OPTIMIZATION

The target code for a compiler may be either an assembly language program or a binary code. Considered here is the case where the target code is an assembly language code.

- control transfer is implemented through conditional and unconditional goto's
- Control structures like *if, for* and *while* introduce semantic gap between the code and the order of execution, because here control transfer is implicit, not explicit. Hence we want to map control structures to program with explicit goto's. For this the compiler generates its own label and put against appropriate statements. Given below is the method to convert code segments with control structures like *if-then-else* and *while* loop converted into intermediate forms. This is also called 'three address code'.

```
if (e) then
        S1;
else
        S2;
S3;
```

```
if (ē) then goto label1;
S1;
goto lable2;
Label1: S2;
Label2: S3;
```

```
while (e) do
        S1;
        S2;
        .
        .
        Sn;
end while;
Sn+1
```

```
if (ē) then goto label2;
S1;
S2;
.
.
Sn;
```

goto lable1;
Label2: S_{n+1};

Function and Procedure calls

Side effect : Change in value of non local variables after a function call.

Actions performed when a procedure or a function call executed are summerised below.

- Actual parameters accessible in called function
- Produce side effects according to scope rule
- Control transferred and returned
- Function value returned
- All other aspects are unaffected.

Function call implementation

- Parameter list : This is a list with descriptor D_p for each parameter.
- Save area : Saves the content of registers before control transferred to called function and restored after return from the called function.
- Calling convention : It answerers the following questions.
    1. How parameter list is accessed?
    2. How save area is accessed?
    3. How control transfer done?
    4. How function value returned?

Calling conventions

Calling convention varies for static memory allocation and dynamic memory allocation.

Static memory:

- Parameter list and save area allocated in calling program
- An actual parameter from parameter list is accessed as
  $D_p = <r_{par\_list}> + (d_{Dp})_{par\_list}$
  Where, r_{par_list} is the register holding the address of the parameter list and hence $<r_{par\_list}>$ denotes the content of r_{par_list} register which is the address of the parameter list and $(d_{Dp})_{par\_list}$ is the displacement to where the descriptor is stored

Dynamic memory:

- Calling function construct parameter list and save area on stack. When execution inititalised, this become part of called functions activation record.
  $D_p = <ARB> + (d_{Dp})_{AR}$
  Where, <ARB> denotes the content of Activation record base pointer and $(d_{Dp})_{AR}$ is the displacement to where the descriptor is stored

Parameter passing

There are basically four differed parameter passing methods. They are given below.

1. Call by value

- Value of actual parameter is copied to the corresponding formal parameter
- On return from the function, the value of the formal parameter is not copied back to corresponding actual parameter. Hence all the changes are local to the called function.

2. Call by value-name
    - Value of actual parameter is copied to the corresponding formal parameter
    - On return from the function, the value of the formal parameter is copied back to corresponding actual parameter.
3. Call by reference
    - Actual parameter address is passed to called function.
    - If the parameter is an expression, it is evaluated and stored at some location and the address of that location is passed to the called function.
4. Call by name
    - Every formal parameter reference inside the called function is replaced by the name of the corresponding actual parameter name. An example for this is given below.

```
int f1(inta, int b)
{
        z = a;
        i = i + 1;
        b = a + 5;
}
```

And the function is called as f1(d[i],x);

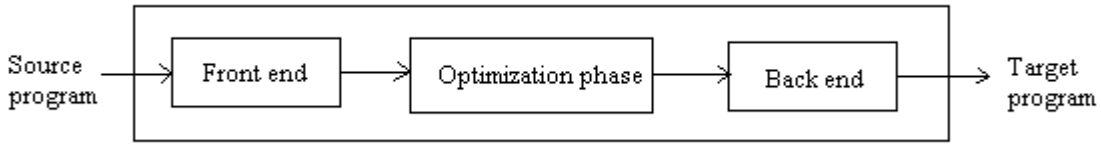
```
int f1(inta, int b)
{
        z = d[i];
        i = i + 1;
        x = d[i] + 5;
}
```

Code Optimization

The aim of this phase is to improve the efficiency of the program code. This is done by

- Eliminating redundancies
- Rewriting program code without changing the basic algorithm for better efficiency.

The block diagram of this phase is as given below.



Optimizing Transformations

These are rules for writing program segment without changing the meaning or algorithm. Some common transformations are given below.

1. Compile time evaluation

   Execution efficiency can be improved by performing certain actions specified in a program during compilation itself. This eliminates the need to perform them during execution of a program, there by reducing the execution time of the program.

   e.g., Constant folding: Operations on constant. For example program code like x = 30/2; can be evaluated during compilation itself.

2. Elimination of common sub expression

   Common sub expressions are occurrence of expression yielding same values.

```
 a = b*c;                          t = b*c;
 --------                           a = t;
 ---------;                        ----------
x=b*c+10;                          ---------
                                    x = t+10
```

3. Dead code elimination

   Codes which can be omitted without affecting the result are called dead codes. e.g., value assigned for a variable and never used there after.

   Eg: while (0)
   {
   x=a+b;
   }

4. Frequency reduction

   Reduce the number of times a program code is executed. e.g., constant assignment inside a loop can be taken outside as given below. Consider the example given below.

```
for (i=0; i<10; i++)                    d = b;
{                                       for (i=0; i<10; i++)
   d = b;                               {
     .                                      .
     .                                      .
}                                       }
```

5. Strength reduction

Replace time consuming operations with faster ones. e.g., multiplication operation can be replaced by repeated addition.

```
Eg: for (i=1; i<5; i++)                 temp=5;
    {                                   for (i=1; i<5; i++)
       k=i*5;                           {
                                           k=temp;
        . . .                           temp=temp+5;
                                          ……
    }                                   }
```

Compiler writing tools

The compiler writer, like any programmer, can profitably use software tools such as debuggers, version managers, profilers, and so on. In addition to the software-development tools, other more specialized tools have been developed for helping implement various phases of a compiler.

Some general tools have been created for the automatic design of specific compiler components. These tools are specialized languages for specifying and implementing the component, and may use algorithms that are quite sophisticated. The most successful tools are those that hide the details of the generation algorithm and produce components that can be easily integrated into the remainder of the compiler. Some useful compiler-construction tools are as follows:

Parser generators: These produce syntax analyzers, normally from input that is based on a CFG. In early compilers, syntax analysis consumed not only a large fraction of the running time of a compiler, but a large fraction of the intellectual effort of writing the compiler. This phase is now considered one of the easiest to implement.

Scanner generators: These automatically generate lexical analyzers, normally from a specification based on regular expressions. The basic organization of the resulting lexical analyzer is in effect, a finite automaton.

Syntax-based translation engines: These produce collections of routines that walk the parse tree, generating intermediate code. The basic idea is that one or more “translations”

are associated with each node of the parse tree, and each translation is defined in terms of translations at its neighbour nodes in the tree.

Automatic code generators: Such a tool takes a collection of rules that define he translation of each operation of the intermediate language into the machine language for the target machine. The rules must include sufficient detail that we can handle the different possible access methods for data (variables may be in registers, in a fixed or static location in memory, or allocated in a stack position.). The basic technique is "template matching". The intermediate code statements are replaced by "templates" that represent sequences of machine instructions, in such a way that the assumptions about storage of variables match from template to template. Since there are usually many options regarding where variables are to be placed (any register or the memory), there are many possible ways to "tile" intermediate code with a given set of templates, and it is necessary to select a good tiling without a combinatorial explosion in running time of the compiler. Often a high-level language especially suitable for specifying the generation of intermediate, assembly, or object code is provided by the compiler-compiler. The user writes routines in this language and, in the resulting compiler, the routines are called at the correct times by the automatically generated parser. A common feature of compiler-compiler is a mechanism for specifying decision tables that select the object code. These tables become part of the generated compiler, along with an interpreter for these tables, supplied by the compiler-compiler.

Data flow engines: Much of the information needed to perform good code optimization involves "data flow analysis", the gathering of information about how values are transmitted from one part of the program to another part. Different tasks of this nature can be performed by essentially the same routine, with the user supplying details of the relationship between intermediate code statements and the information being gathered.

Lex

Lex is a program generator designed for lexical processing of character input streams. It accepts a high-level, problem oriented specification for character string matching, and produces a program in a general purpose language which recognizes regular expressions. The regular expressions are specified by the user in the source specifications given to Lex. The Lex written code recognizes these expressions in an input stream and partitions the input stream into strings matching the expressions. At the boundaries between strings program sections provided by the user are executed. The Lex source file associates the regular expressions and the program fragments. As each expression appears in the input to the program written by Lex, the corresponding fragment is executed.

Lex is not a complete language, but rather a generator representing a new language feature which can be added to different programming languages, called ``host languages." Just as general purpose languages can produce code to run on different computer hardware, Lex can write code in different host languages. The host language is used for the output code generated by Lex and also for the program fragments added by the user. Compatible run-time libraries for the different host

languages are also provided. This makes Lex adaptable to different environments and different users. Input to Lex is divided into three sections, with **%%** dividing the sections. The structure of a lex file is as given below.

... definitions ...
%%
... rules ...
%%
... subroutines ...

Yacc

Yacc is a piece of computer software that serves as the standard parser generator on Unix systems. The name is an acronym for "Yet Another Compiler Compiler." It generates a parser (the part of a compiler that tries to make sense of the input) based on an analytic grammar written in BNF notation. Yacc generates the code for the parser in the C programming language. It was developed by Stephen C. Johnson at AT&T for the Unix operating system.

Since the parser generated by Yacc requires a lexical analyzer, it is often used in combination with a lexical analyzer generator, in most cases either Lex or the free software alternative Flex. The structure of a Yacc file is as given below.

... definitions ...
%%
... rules ...
%%
... subroutines ...

Backpatching

The problem of forward reference is tackled using a process called backpatching. The operand field of an instruction containing a forward reference is left blank initially. The address of the forward reference symbol is put in to this field when its definition is encountered.