COMPILER DESIGN - CODE GENERATION

Code generation can be considered as the final phase of compilation. Through post code generation, optimization process can be applied on the code, but that can be seen as a part of code generation phase itself. The code generated by the compiler is an object code of some lower-level programming language, for example, assembly language. We have seen that the source code written in a higher-level language is transformed into a lower-level language that results in a lower-level object code, which should have the following minimum properties:

- It should carry the exact meaning of the source code.
- It should be efficient in terms of CPU usage and memory management.

We will now see how the intermediate code is transformed into target object code *assemblycode, inthiscase*.

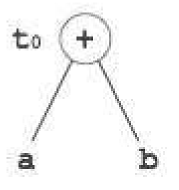
Directed Acyclic Graph

Directed Acyclic Graph *DAG* is a tool that depicts the structure of basic blocks, helps to see the flow of values flowing among the basic blocks, and offers optimization too. DAG provides easy transformation on basic blocks. DAG can be understood here:

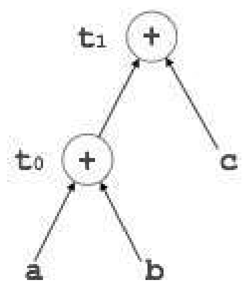
- Leaf nodes represent identifiers, names or constants.
- Interior nodes represent operators.
- Interior nodes also represent the results of expressions or the identifiers/name where the values are to be stored or assigned.

Example:

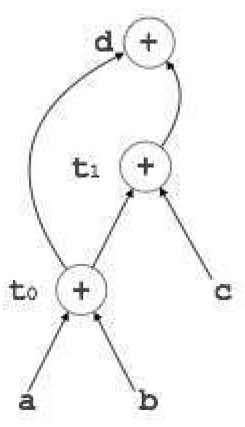
```
t0 = a + b
t1 = t0 + c
d = t0 + t1
```



[$t_0 = a + b$]



[$t_1 = t_0 + c$]



[$d = t_0 + t_1$]

Peephole Optimization

This optimization technique works locally on the source code to transform it into an optimized code. By locally, we mean a small portion of the code block at hand. These methods can be applied on intermediate codes as well as on target codes. A bunch of statements is analyzed and are checked for the following possible optimization:

Redundant instruction elimination

At source code level, the following can be done by the user:

```
int add_ten(int x)
   {
   int y, z;
   y = 10;
   z = x + y;
   return z;
   }
```

```
int add_ten(int x)
   {
   int y;
   y = 10;
   y = x + y;
   return y;
   }
```

```
int add_ten(int x)
   {
   int y = 10;
   return x + y;
   }
```

```
int add_ten(int x)
   {
   return x + 10;
   }
```

At compilation level, the compiler searches for instructions redundant in nature. Multiple loading and storing of instructions may carry the same meaning even if some of them are removed. For example:

- MOV x, R0
- MOV R0, R1

We can delete the first instruction and re-write the sentence as:

```
MOV x, R1
```

Unreachable code

Unreachable code is a part of the program code that is never accessed because of programming constructs. Programmers may have accidently written a piece of code that can never be reached.

Example:

```
void add_ten(int x)
{
   return x + 10;
   printf("value of x is %d", x);
}
```

In this code segment, the **printf** statement will never be executed as the program control returns back before it can execute, hence **printf** can be removed.

Flow of control optimization

There are instances in a code where the program control jumps back and forth without performing any significant task. These jumps can be removed. Consider the following chunk of code:

```
...
MOV R1, R2
GOTO L1
...
L1 :   GOTO L2
L2 :   INC R1
```

In this code,label L1 can be removed as it passes the control to L2. So instead of jumping to L1 and then to L2, the control can directly reach L2, as shown below:

```
...
MOV R1, R2
GOTO L2
...
L2 :   INC R1
```

Algebraic expression simplification

There are occasions where algebraic expressions can be made simple. For example, the expression **a = a + 0** can be replaced by **a** itself and the expression a = a + 1 can simply be replaced by INC a.

Strength reduction

There are operations that consume more time and space. Their ‘strength’ can be reduced by replacing them with other operations that consume less time and space, but produce the same result.

For example, **x * 2** can be replaced by **x << 1**, which involves only one left shift. Though the output of a * a and a^2 is same, a^2 is much more efficient to implement.

Accessing machine instructions

The target machine can deploy more sophisticated instructions, which can have the capability to perform specific operations much efficiently. If the target code can accommodate those instructions directly, that will not only improve the quality of code, but also yield more efficient results.

Code Generator

A code generator is expected to have an understanding of the target machine’s runtime environment and its instruction set. The code generator should take the following things into consideration to generate the code:

- **Target language** : The code generator has to be aware of the nature of the target language for which the code is to be transformed. That language may facilitate some machine-specific instructions to help the compiler generate the code in a more convenient way. The target machine can have either CISC or RISC processor architecture.

- **IR Type** : Intermediate representation has various forms. It can be in Abstract Syntax Tree *AST* structure, Reverse Polish Notation, or 3-address code.

- **Selection of instruction** : The code generator takes Intermediate Representation as input and converts *maps* it into target machine’s instruction set. One representation can have many ways *instructions* to convert it, so it becomes the responsibility of the code generator to choose the appropriate instructions wisely.

- **Register allocation** : A program has a number of values to be maintained during the execution. The target machine’s architecture may not allow all of the values to be kept in the CPU memory or registers. Code generator decides what values to keep in the registers. Also, it decides the registers to be used to keep these values.

- **Ordering of instructions** : At last, the code generator decides the order in which the instruction will be executed. It creates schedules for instructions to execute them.

Descriptors

The code generator has to track both the registers *foravailability* and addresses *locationofvalues* while generating the code. For both of them, the following two descriptors are used:

- **Register descriptor** : Register descriptor is used to inform the code generator about the availability of registers. Register descriptor keeps track of values stored in each register. Whenever a new register is required during code generation, this descriptor is consulted for register availability.

- **Address descriptor** : Values of the names *identifiers* used in the program might be stored at different locations while in execution. Address descriptors are used to keep track of memory locations where the values of identifiers are stored. These locations may include CPU registers, heaps, stacks, memory or a combination of the mentioned locations.

Code generator keeps both the descriptor updated in real-time. For a load statement, LD R1, x, the code generator:

- updates the Register Descriptor R1 that has value of x and
- updates the Address Descriptor x to show that one instance of x is in R1.

Code Generation

Basic blocks comprise of a sequence of three-address instructions. Code generator takes these sequence of instructions as input.

Note : If the value of a name is found at more than one place $register, cache, ormemory$, the register’s value will be preferred over the cache and main memory. Likewise cache’s value will be preferred over the main memory. Main memory is barely given any preference.

getReg : Code generator uses *getReg* function to determine the status of available registers and the location of name values. *getReg* works as follows:

- If variable Y is already in register R, it uses that register.
- Else if some register R is available, it uses that register.
- Else if both the above options are not possible, it chooses a register that requires minimal number of load and store instructions.

For an instruction x = y OP z, the code generator may perform the following actions. Let us assume that L is the location $preferablyregister$ where the output of y OP z is to be saved:

- Call function getReg, to decide the location of L.
- Determine the present location $registerormemory$ of **y** by consulting the Address Descriptor of **y**. If **y** is not presently in register **L**, then generate the following instruction to copy the value of **y** to **L**:

  MOV y’, L

  where **y’** represents the copied value of **y**.
- Determine the present location of **z** using the same method used in step 2 for **y** and generate the following instruction:

  OP z’, L

  where **z’** represents the copied value of **z**.
- Now L contains the value of y OP z, that is intended to be assigned to **x**. So, if L is a register, update its descriptor to indicate that it contains the value of **x**. Update the descriptor of **x** to indicate that it is stored at location **L**.
- If y and z has no further use, they can be given back to the system.

Other code constructs like loops and conditional statements are transformed into assembly language in general assembly way.