## Code Optimization

In recent years, most research and development in the area of compiler design has been focused on the optimization phases of the compiler. **Optimization** is the process of improving generated code so as to reduce its potential running time and/or reduce the space required to store it in memory. Software designers are often faced with decisions which involve a space-time tradeoff – i.e., one method will result in a faster program, another method will result in a program which requires less memory, **but no method will do both**. However, many **optimization techniques** are capable of improving the **object program** in both time and space, which is why they are employed in most modern compilers. This results from either the fact that much effort has been directed toward the development of optimization techniques, or from the fact that the code normally generated is very poor and easily improved.

Optimization techniques can be **separated into two general classes**: local and global.

- **Local optimization techniques** normally are concerned with transformations on small sections of code (involving only a few instructions) and generally operate on the machine language instructions which are produced by the code generator.
- **Global optimization techniques** are generally concerned with larger blocks of code, or even multiple blocks or modules, and will be applied to the intermediate form, atom strings, or syntax trees put out by the parser.

## Peephole Optimization

Code generated by using the statement-by-statement code-generation strategy contains redundant instructions and suboptimal constructs. Therefore, to improve the quality of the target code, optimization is required. **Peephole optimization** is an effective technique for locally improving the target code. Short sequences of target code instructions are examined and replacement by faster sequences wherever possible. Typical optimizations that can be performed are:

- Elimination of redundant loads and stores
- Elimination of multiple jumps
- Elimination of unreachable code
- Algebraic simplifications
- Reducing for strength
- Use of machine idioms

**Eliminating Redundant Loads and Stores**

If the target code contains the instruction sequence:

1. **MOV R, *a***
2. **MOV *a*, R**

We can delete the **second instruction** if it an unlabeled instruction. This is because the first instruction ensures that the value of ***a*** is already in the register **R**. If it is labeled, there is no guarantee that **step 1** will always be executed before step 2.

**Eliminating Multiple Jumps**

If we have jumps to other jumps, then the unnecessary jumps can be eliminated in either intermediate code or the target code. If we have a jump sequence:

```
goto  L 1
...
L 1:   goto  L 2
```

**Then this can be replaced by:**

```
      goto  L 2
      ...
L 1:   goto  L 2
```

If there are now no jumps to L1, then it may be possible to eliminate the statement, provided it is preceded by an unconditional jump. Similarly, the sequence:

```
if  a  <  b  goto  L 1
      ...
L 1:   goto  L 2
```

can be replaced by:

```
if  a  <  b  goto  L 2
      ...
L 1:   goto  L 2
```

**Eliminating Unreachable Code**

In computer programming, **unreachable code** is part of the source code of a program which can never be executed because there exists no control flow path to the code from the rest of the program.

Unreachable code is sometimes also called dead code, although dead code may also refer to code that is executed but has no effect on the output of a program.

Consider the following fragment of C code:

```
int foo (int iX, int iY)
{
 return iX + iY;
```

```
  int iZ = iX*iY;
 }
```
The definition int **iZ = iX*iY**; is never reached as the function returns before the definition is reached. Therefore the definition of **iZ** can be discarded.

## Algebraic Simplifications

If statements like:
**a= a+0**
**a=a*1**

are generated in the code, they can be eliminated, because zero is an additive identity, and one is a multiplicative identity.

## Reducing Strength

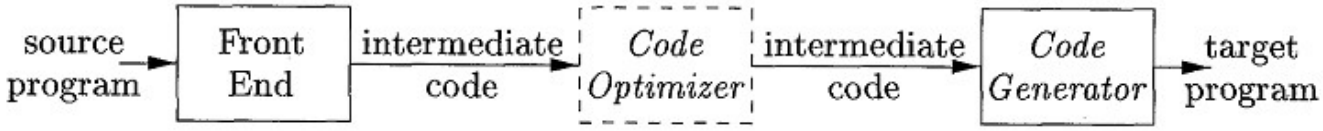
Certain machine instructions are considered to be cheaper than others. Hence, if we replace expensive operations by equivalent cheaper ones on the target machine, then the efficiency will be better.
**For example, $x^2$**
is invariable cheaper to implement as **x * x** than as a call to an exponentiation routine. Similarly, **fixed-point multiplication** or **division by a power of two is cheaper to implement as a shift**.

## Using Machine Idioms

The target machine may have hardware instructions to implement certain specific operations efficiently. Detecting situations that permit the use of these instructions can reduce execution time significantly. **For example**, some machines have auto-increment and auto-decrement addressing modes. Using these modes can greatly improve the quality of the code when **pushing or popping a stack**. These modes can also be used for implementing statements like
**a = a + 1**.
**Example:**
Post increment or pre-decrement addressing
**r++ , --r**

## Code Generation

The final phase in compiler model is the code generator. It takes as input the **intermediate representation (IR)** produced by the front end of the compiler, along with relevant symbol table information, and produces as output a semantically equivalent target program, as shown in figure below.




**Position of Code Generator**

The requirements imposed on a code generator are severe. The target program must preserve the **semantic meaning of the source program and be of high quality**; that is, it must make **effective use of the available resources** of the target machine. Moreover, the code generator itself must run efficiently.
The challenge is that, mathematically, the problem of generating an optimal target program for a given source program is undecidable; many of the subproblems encountered in code generation such as **register allocation** are computationally intractable. In practice, we must be content with heuristic techniques that generate good, but not necessarily optimal, code. Fortunately, heuristics have matured enough that a carefully designed code generator can produce code that is several times faster than code produced by a naive one. Compilers that need to produce efficient target programs, **include an optimization phase prior to code generation**. The optimizer **maps the IR into IR from** which more efficient code can be generated. In general, the **code optimization and code-generation phases** of a compiler, often referred to as the **back end**, may make multiple passes over the IR before generating the target program. The most important criterion for a code generator is that it produces correct code.

## 1. Input to the Code Generator

The input to the code generator is

- The intermediate representation of the source program produced by the front end,
- The information in the **symbol table** that is used to determine the run-time addresses of the data objects denoted by the names in the **IR**.

The many choices for the **IR** include three-address representations such as quadruples, triples, indirect triples; **virtual machine** representations such as **bytecodes and stack-machine code**; **linear representations** such as postfix notation; and **graphical representations** such as syntax trees and DAG's.

Code generator main tasks:
- Instruction selection
- Register allocation and assignment
- Instruction ordering

## Instruction Selection

The code generator must map the **IR** program into a **code sequence** that can be executed by the target machine. The complexity of performing this mapping is determined by factors such as
- The level of the IR
- The nature of the instruction-set architecture
- The desired quality of the generated code.

- If the **IR is high level**, the code generator may translate each **IR** statement into a sequence of machine instructions using **code templates**. Such statement-by-statement code generation, however, often produces poor code that needs further **optimization**. If the **IR reflects some of the low-level** details of the underlying machine, then the code generator can use this information to generate more **efficient code sequences**.

- The **nature of the instruction set** of the target machine has a strong effect on the difficulty of instruction selection. **For example**, the uniformity and completeness of the instruction set are important factors. If the **target machine does not support each data type** in a uniform manner, then each exception to the general rule requires special handling. On some machines, **for example**, **floating-point operations are done using separate registers**.

Instruction speeds and machine idioms are other important factors. If we do not care about the efficiency of the target program, instruction selection is straightforward. For each type of three-address statement, we can design a code skeleton that defines the target code to be generated for that construct.

**For example**, every **three-address statement** of the form **x = y + z**, where x, y, and z are statically allocated, can be translated into the code sequence

```
LD  R0, y          // R0  =  y  (load y into register R0)
ADD R0, R0, z      //  R0  = R0  +  z  (add z to R0 )
ST  x,  R0         // x  =  R0  (store R0  into x)
```

This strategy often produces redundant loads and stores. **For example**, the Sequence of three-address statements

**a = b + c**
**d = a + e**

would be translated into

```
LD  R0, b          // R0  =  b
ADD  R0, R0,  c    // R0  =  R0  +  c
ST  a, R0          // a  =  R0
LD  R0,  a         // R0  =  a
ADD  R0, R0, e     // R0  =  R0  +  e
ST  d,  R0         // d  =  R0
```

Here, the **fourth statement is redundant** since it loads a value that has just been stored, and so is the third if **a** is not subsequently used.

➢ The **quality of the generated code** is usually determined by its **speed and size**. On most machines, a given **IR** program can be implemented by many different code sequences, with significant cost differences between the different implementations. A naive translation of the intermediate code may therefore lead to correct but unacceptably inefficient target code.
**For example**, if the target machine has an "increment" instruction (INC), then the three-address statement **a = a + 1** may be implemented more efficiently by the single instruction INC a, rather than by a more obvious sequence that loads **a** into a register, adds one to the register, and then stores the result back into a:

```
LD  R0, a          // R0 =  a
ADD R0,  R0,  1    // R0 =  R0 +  1
ST  a, R0          // a =  R0
```

## Register Allocation

A key problem in code generation is deciding **what values to hold in what registers**. **Registers are the fastest computational unit on the target machine**, but we usually do not have enough of them to hold all values. **Values not held in registers need to reside in memory**. Instructions involving register operands are invariably shorter and faster than those involving operands in memory, **so efficient utilization of registers is particularly important**.
**The use of registers is often subdivided into two subproblems**:
1. **Register allocation**, during which we select the set of variables that will reside in registers at each point in the program.
2. **Register assignment**, during which we pick the specific register that a variable will reside in.

## Evaluation Order

The order in which computations are performed can affect the efficiency of the target code. **Some computation orders require fewer registers to hold intermediate** results than others. Initially, we shall avoid the problem by generating code for the three-address statements in the order in which they have been produced by the intermediate code generator.