Chapter 8
Code Generation

2022 Spring&Summer

8.5 Code Generation of Procedure and Function Calls

- Different target machines use considerably different mechanisms for performing calls and calls depend heavily on the organization of the runtime environment.
- It is difficult to achieve an intermediate code representation that is general enough to use for any target architecture or runtime environment.

8.5.1 Intermediate Code for Procedures and Functions

- First, there are actually two mechanisms that need descriptions: function/procedure definition and function/procedure call.
- A definition creates a function name, parameters, and code, but the function does not execute at that point.
- A call creates actual values for the parameters and performs a jump to the code of the function, which then executes and returns.

8.5.1 Intermediate Code for Procedures and Functions

- Intermediate code for a definition must include an instruction marking the beginning, or entry point, of the code for the function, and an instruction marking the ending, or return point, of the function.
- We can write this as follows:

Entry instruction

<code for the function body>

Return instruction

8.5.1 Intermediate Code for Procedures and Functions

- Similarly, a function call must have an instruction indicating the beginning of the computation of the arguments and the an actual call instruction that indicates the point where the arguments have been constructed and the actual jump to the code of the function can take place:

Begin-argument-computation instruction

<code to compute the arguments >

Call instruction

8.5.1 Intermediate Code for Procedures and Functions

1) Three-Address Code for Procedures and Functions

In three-address code, the entry instruction needs to give a name to the procedure entry point, similar to the label instruction; thus, it is a one-address instruction, which we will call simply entry. Similarly, we will call the return instruction return.

For example, consider the C function definition.

int f (int x, int y)

{ return x + y + 1; }

This will translate into the following three-address code:

entry f

t1 = x + y

t2 = t1 + 1

return t2

8.5.1 Intermediate Code for Procedures and Functions

Three different three-address instructions are needed:

1. One to signal the start of argument computation, which we will call *begin_args* (and which is a zero-address instruction);

2. An instruction that is used repeatedly to specify the names of argument values, which we will call *arg* (and which must include the address , or name, of the argument value)

3. The actual *call* instruction, which we will write simply as call, which also is a one-address instruction (the name or entry point of the function being called must be given).

8.5.1 Intermediate Code for Procedures and Functions

For example, suppose the function f has been defined in C as in the previous example. Then, the call

f (2+3, 4)

translates to the three-address code

begin_args

t1 = 2 + 3

arg t1

arg 4

call f

8.5.1 Intermediate Code for Procedures and Functions

2) P_code for Procedures and functions

The entry instruction in P-code is ent, and the return instruction is ret. Thus the previous definition of the C function f translates into the P-code

ent f

lod x

lod y

adi

ldc 1

adi

ret (the ret instruction does not need a parameter to indicate what the returned value is assumed to be at top of the P-machine stack on return.)

8.5.1 Intermediate Code for Procedures and Functions

P-code instructions for a call are the *mst* instruction and the *cup* instruction.

- *mst* instruction: stands for "mark stack" and corresponds to the *begins_args* , the target code generated from such an instruction is concerned with setting up the activation record for the new call on the stack. That space must be allocated or "marked" on the stack for such items as the arguments.
- *cup* instruction: is the "*call user procedure*" instruction and corresponds directly to the call instruction of three-address code.
- All argument values are assumed to appear on the stack (in the appropriate order) when the *cup* instruction is encountered.

8.5.1 Intermediate Code for Procedures and Functions

Our example of a call f (2+3, 4) now translates into the following P-code:

mst

ldc 2

ldc 3

adi

ldc 4

cup f

8.5.2 A Code Generation Procedure for Function Definition and Call

The grammar we will use is the following:

- *program* → *decl-list exp*
- *decl-list* → *decl-list decl* | ε
- *decl* → fn *id (param-list)* = *exp*
- *param-list* → *param-list, id* | *id*
- *exp* → *exp* + *exp* | *call* | *num* | *id*
- *call* → *id (arg-list)*
- *arg-list* → *arg-list, exp* | *exp*

all values are integers, all functions return integers, and all functions must have at least one parameter.

8.5.2 A Code Generation Procedure for Function Definition and Call

An example of a program as defined by this grammar is:

fn f(x) = 2+x

fn g(x,y) = f(x)+y

g(3,4)

8.5.2 A Code Generation Procedure for Function Definition and Call

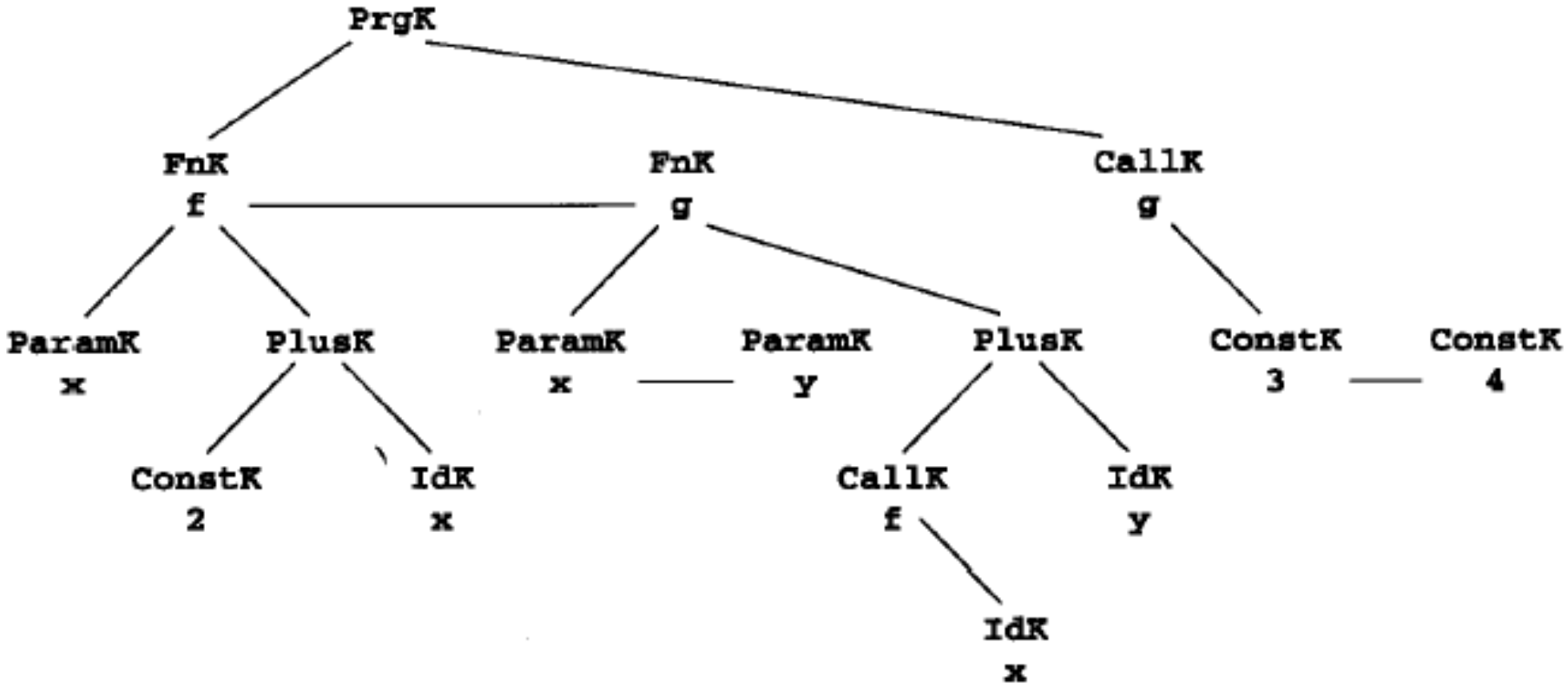
```
typedef enum
{PrgK, FnK, ParamK, PlusK, CallK, ConstK, IdK}
                                              NodeKind ;
typedef struct streenode
        { NodeKind kind;
      struct streenode  * lchild,  * rchild,  * sibling ;
        char * name;            /* used with
                                   FnK,ParamK,Callk,IdK */
        int val; /* used with ConstK */
       } StreeNode;
typedef StreeNode * SyntaxTree;
```

8.5.2 A Code Generation Procedure for Function Definition and Call

Each syntax tree has a root that is a *PrgK* node. This node is used simply to bind the declarations and the expression of the program together. The left child of this node is a sibling list of *FnK* nodes. The right child is the associated program expression.

Each FnK node has a left child that is a sibling list of *ParamK* nodes.

A *CallK* node contains the name of the called function and has a right child that is a sibling list of the argument expressions.


PrgK
FnK
f
FnK
g
CallK
g
ParamK
x
PlusK
ParamK
x
ParamK
y
PlusK
ConstK
3
ConstK
4
ConstK
2
IdK
x
CallK
f
IdK
y
IdK
x

8.5.2 A Code Generation Procedure for Function Definition and Call

- fn f(x) = 2+x
- fn g(x,y) = f(x)+y
- g(3,4)

```
ent f
 ldc 2
 lod x
 adi
 ret
 ent g
 mst
 lod x
 cup f
 lod y
 adi
 ret
 mst
 ldc 3
 ldc 4
 cup g
```

8.6 Code Generation in Commercial Compilers: Two Case Studies- The Borland 3.0 C Compiler for the 80X86

Example: (x = x +3) + 4

- mov ax, word ptr [bp-2]
- add ax, 3
- mov word ptr [bp-2], ax
- add ax, 4

The accumulator register ax is used as the main temporary location for the computation.

The location of the local variable *x* is *bp-2*, reflecting the use of the register *bp* (base pointer) as the frame pointer and the fact that integer variables occupy two bytes on this machine.

The Borland 3.0 C Compiler for the 80X86

1) Array Reference

A example: *(a [i + 1] = 2) + a [j]*

Assume that i j, and a are local variables declared as

int i,j;

int a[10];

the following assembly code for the expression:

(1) mov bx,word ptr [bp-2]

(2) shl bx , 1

(3) lea ax, word ptr [bp-22] load the address

(4) add bx , ax computes the resulting address of a[i+1] into bx

(5) mov ax , 2

(6) mov word ptr [bx],ax

(7) mov bx,word ptr [bp-4]

(8) shl bx , 1

(9) lea dx,word ptr [bp-24]

(1 0) add bx , dx

(1 1) add ax,word ptr [bx] the resulting value of the expression is left in register ax

The Borland 3.0 C Compiler for the 80X86

bp-2 is the location of *i* in the local activation record, *bp-4* is the location of *j*, the base address of *a* is *bp-24*

address (a [i + 1])

*= base _ address (a) + (i + 1)*elem_size (a)*

*= (base _ address (a) + elem_size (a)) + i*elem_size (a)*

The Borland 3.0 C Compiler for the 80X86

2) Pointer and Field References

assume the declarations of previous examples:

```
typedef struct rec
  { int i;
  char c;
  int j;
  } Rec;
typedef struct treeNode
{ int val;
struct treeNode * lchild, * rchild;
} TreeNode;
…
Rec x;
TreeNode *p;
```

The Borland 3.0 C Compiler for the 80X86

- The sizes of the data types : Integer variables have size 2 bytes, character variables have size 1 byte, and pointers are so-called “near” pointers with 2 bytes.
- Variable *x* has size 5 bytes and the variable *p* is 2 bytes. *x* is allocated in the activation record at location *bp-6* (local variables are allocated only on even-byte boundaries). And ***p*** is allocated to register ***si***.

The Borland 3.0 C Compiler for the 80X86

The code generated for the statement:

- **x.j =x.i;**

is

- **mov ax,word ptr [bp-6]**
- **mov word ptr [bp-3],ax**

Note how the offset computation for j (-6 + 3 = -3) is performed statically by the compiler.

- **The code generated for the statement**
- **p->lchild = p;**

is

- **mov word ptr [si+2], si**
- **and the statement p = p->rchild;**

is

- **mov si, word ptr [si+4]**

The Borland 3.0 C Compiler for the 80X86

3) *If and While-Statement*

The statements we use are

- **if (x>y) y++;else x--;**

and

- **while (x<y) y -= x;**

The Borland compiler generates the following 80x86 code for the given if-statement: *x* is located in register ***bx*** and ***y*** is located in register ***dx***.

The Borland 3.0 C Compiler for the 80X86

if (x>y) y++;else x--;

```
            cmp bx , dx
            jle short @1@86
            inc dx
            jmp short @1@114
@1@86 :
           dec bx
@1@114 :
```

The Borland 3.0 C Compiler for the 80X86

while (x<y) y -= x;

```
            jmp short @1@170
@1@142 :
            sub dx , bx
@1@170 :
            cmp bx , dx
            jl short @1@142
```

The test is placed at the end, and an initial unconditional jump is made to this test.

The Borland 3.0 C Compiler for the 80X86

4) function definition and call

We will use are the C function definition:

int f(int x, int y)

{ return x+y+1 ; }

and a corresponding call

f (2+3, 4)

the Borland compiler for the call f (2+3, 4):

```
mov ax,4
push ax
mov ax,5
push ax
call near ptr    _f
pop cx
pop cx
```

The Borland 3.0 C Compiler for the 80X86

- Note how the arguments are pushed on the stack in reverse order. Note also that the caller is responsible for removing the arguments from the stack after the call. This is the reason for the two **POP** instructions after the call).
- **The near ptr** declaration means that the function is in the same segment.
- The **call** instruction on the 80x86 automatically pushes the return address on the stack. A corresponding **ret** instruction executed by the called function will pop it.

The Borland 3.0 C Compiler for the 80X86

```
_ f  proc near
     push bp            save the control link ( bp ) on the stack
     mov bp , sp        set the bp to the current sp
     mov ax,word ptr [bp+4]
     add ax,word ptr [bp+6]
      inc ax            the returned value is left in register ax
      jmp short @1@58
@1@58 :          since return statements embedded in the function
code
     pop bp
     ret
_ f       endp
```

8.7 TM: A Simple Target Machine

- We generate target code directly for a very simple machine that can be easily simulated. We call this machine TM (for Tiny Machine).

8.7.1 Basic Architecture of the Tiny Machine

- TM consists of a read-only instruction memory, a data memory, and a set of eight general-purpose registers. These all use non-negative integer addresses beginning at 0. Register 7 is the program counter and is the only special register.

8.7.1 Basic Architecture of the Tiny Machine

- #define IADDR_SIZE ...
- /* size of instruction memory */
- #define DADDR_SIZE……
- /* size of data memory */
- #define NO_REGS 8 /* number of registers */
- #define PC_REG 7

- Instruction iMem[IADDR_SIZE];
- int dMem[DADDR_SIZE];
- int reg[NO_REGS];

8.7.1 Basic Architecture of the Tiny Machine

TM performs a conventional fetch-execute cycle:

- **do**
- **/* fetch */**
- **current Instruction = iMem [reg[pcRegNo]++];**
- **/* execute current instruction */**
- **. . .**
- **while (!(halt||error));**

At start up, the Tiny Machine sets all registers and data memory to 0, then loads the value of the highest legal address (namely DADDR_SIZE –1) into dMem[0]. The TM then starts to execute instructions beginning at iMem[0]. The machine stops when a HALT instruction is executed.

The possible error conditions include IMEM_ERR, which occurs if reg[PC_REG] < 0 or reg[PC_REG] >= IADDR_SIZE in the fetch step above.

8.7.1 Basic Architecture of the Tiny Machine

- A register-only instruction has the format **opcode r, s, t**

- A register-memory instruction has the format **opcode r, d(s)**

8.7.1 Basic Architecture of the Tiny Machine

Format *opcode r,s,t*

- Opcode Effect
- HALT stop execution (operands ignored)
- IN reg [r] ← integer value read from the standard input (s and t ignored)
- OUT reg [r] → the standard output (s and t ignored)
- ADD reg [r] = reg[s] + reg[t]
- SUB reg [r] = reg[s] - reg[t]
- MUL reg [r] = reg[s] * reg[t]
- DIV reg [r] = reg[s] / reg[t] (may generate ZERO _ DIV)

8.7.1 Basic Architecture of the Tiny Machine

Format *opcode r,d(s)*

- (a=d+ reg[s]; any reference to DMem [a] generates DEME_ERR if a<0 or a≥DADDR – SIZE)
- Opcode Effect
- LD reg [r] = dMem[a] (load r with memory value at a)
- LDA reg [r] = a (load address a directly into r)
- LDC reg [r] = d (load constant d directly into r – s is ignored)
- ST dMem[a] = reg[r] (store value in r to memory location a)
- JLT if (reg [r]<0) reg [PC_REG] = a
- (jump to instruction a if r is negative, similarly for the following)
- JLE if (reg [r]<=0 reg) [PC_REG] = a
- JGE if (reg [r]>=0) reg) [PC_REG] = a
- JGT if (reg [r]>0) reg) [PC_REG] = a
- JEQ if (reg [r]==0) reg [PC_REG] = a
- JNE if (reg [r]!=0) reg [PC_REG] = a

8.7.1 Basic Architecture of the Tiny Machine

- The target register in the arithmetic, IN, and load operations comes first, and the source register(s) come second. There is no restriction on the use of registers for sources and target; in particular, the source and target registers may be the same.
- All arithmetic operations are restricted to registers.
- There are no floating-point operations or floating-point registers.
- There are no addressing modes specifiable in the operands as in some assembly code. Indeed, the TM has very few addressing choices.
- There is no restriction on the use of the pc in any of the instructions, It must be simulated by using the pc as the target register in an LDA instruction:
  - Example: LDA 7, d(s)
  - This instruction has the effect of jumping to location a = d + reg[s]

8.7.1 Basic Architecture of the Tiny Machine

- There is also no indirect jump instruction, but it too can be imitated, if necessary, by using an LD instruction.
  - LD 7, 0(1) jumps to the instruction whose address is stored in memory at the location pointed to by register 1.
- The conditional jump instructions (JLT, etc.) can be made relative to the current position in the program by using the pc as the second register.
- JEQ 0, 4(7) causes the TM to jump five instructions forward in the code if register 0 is 0.
- There is no procedure call or JSUB instruction.
  - LD 7, d(s)
  - Jumping to the procedure whose entry address is dMem[d+reg[s]].
  - Saving the return address first by executing something like
  - LDA 0, 1(7) (reg[0] is where we want to return to, assuming the next instruction is the actual jump to the procedure).

8.7.2 The TM Simulator

The machine simulator accepts text files containing TM instructions as described above, with the following conventions:

- An entirely blank line is ignored.
- A line beginning with an asterisk is considered a comment and is ignored.
- Any other line must contain an integer instruction location followed by a colon followed by a legal instruction. Any text occurring after the instruction is considered a comment and is ignored.

8.7.2 The TM Simulator

```
* This program inputs an integer, computes
* its factorial if it is positive,
* and prints the result
0 :    IN 0, 0, 0                  r0 = read
1 :    JLE 0, 6 (7)       if 0 < r0 then
2 :    LDC 1,1,0                   r1 = 1
3 :    LDC 2, 1, 0                 r2 = 1
                          * repeat
4 :   MUL 1, 1, 0                  r1 = r1*r0
5 :   SUB 0, 0, 2                  r0 = r0-r2
6 :   JNE 0, -3 (7)      until r0 == 0
7 :   OUT 1, 0, 0                  write r1
8 :   HALT 0, 0, 0       halt
* end of program
```

8.7.2 The TM Simulator

A code generator is likely to generate the code of Figure 8.16 in the following sequence:

- 0 : IN 0,0,0
- 2 : LDC 1,1,0
- 3 : LDC 2,1,0
- 4 : MUL 1,1,0
- 5 : SUB 0,0,2
- 6 : JNE 0,-3(7)
- 7 : OUT 1,0,0
- 1 : JLE 0,6(7)
- 8 : HALT 0,0,0

8.8 Code Generation for the Tiny Language

- The interface of the TINY code generator to the TM, together with the utility functions necessary for code generation.
- Describe the steps of the code generator proper.
- Describe the use of the TINY compiler in combination with the TM simulator.
- Discuss the target code file for the sample TINY program used throughout the book.

8.8.1 The TM Interface of the TINY Code Generator

- Encapsulate some of the information the code generator needs to know about the TM in files code.h and code.c, which are listed in Appendix b.
- Assigns locations at the top of data memory to temporaries and locations at the bottom of data memory to variables. The locations for variables and temporaries can be viewed as absolute addresses.
- The mp will be used for memory access to temporaries and will always contain the highest legal memory location, while the gp will be used for all named variable memory accesses and will always contain 0.

8.8.1 The TM Interface of the TINY Code Generator

- The other two registers that will be used by the code generator are registers 0 and 1, which are called “accumulators” and given the names ac and ac1. In particular, the results of computations will always be left in the ac.

8.8.1 The TM Interface of the TINY Code Generator

- There are seven code emitting functions:
- EmitComment: prints its parameter string in the format of a comment on a separate line in the code file, if the TraceCode flag is set.
- emitRO emitRM: the standard code emitting functions that are used for the R0 and RM classes of instructions, respectively.
- emitSkip : skip a number of locations that will later be backpatched.
- emitBackup: set the current instruction location to a previous location for backpatching.
- emitRestore: return the current instruction location to the value prior to a call to emitBackup.
- emitRM_Abs: to generate the code for such a backpatched jump or any jump to a code location returned by a call to emitSkip.

8.8.2 The TINY Code Generator

- The TINY code generator is contained in file cgen.c, with its only interface to the TINY compiler the function codeGen, with prototype

- **void codeGen(void);**

- The **codeGen** function does: it generates a few comments and instructions that set up the runtime environment on startup, the calls the **cGen** function on the syntax tree, and finally generates a **HALT** instruction to end the program.

8.8.2 The TINY Code Generator

- The cGen function tests only whether a node is a statement or expression node(or null), calling the appropriate function genStmt or genExp, and then calling itself recursively on siblings.
- The genStmt function contains a large **switch** statement that distinguishes among the five kinds of statements, generating appropriate code and recursive calls to **cGen** in each case, and similarly for the **genExp** function.
- In all cases, the code for a subexpression is assumed to leave a value in the **ac**.

8.9 A Survey of Code Optimizations Techniques

- The quality of the code can be measured by the speed and the size of the target code.
- It is important to realize that a compiler writer cannot hope to include every single optimization technique that has ever been developed.
- techniques are most likely to result in a ***significant code improvement*** with ***the smallest increase*** in the complexity of the compiler.

8.9.1 Principal Sources of Code Optimizations

The areas in which a code generator may fail to produce good code:

1) Register Allocation :Good use of registers is the most important feature of efficient code. Historically, the number of available registers was severely limited.

- One approach: increase the number and speed of operations that can be performed directly in memory.
- Another approach: (the so-called RISC approach) has been to decrease the number of operations that can be performed directly in memory (often to zero), but at the same time to increase the number of available registers to 32, 64 or 128. In such architectures proper register allocation becomes even more critical.

8.9.1 Principal Sources of Code Optimizations

2) Unnecessary Operations:

The second major source of code improvement is to avoid generating code for operations that are redundant or unnecessary.

This kind of optimization can vary from a very simple search of localized code to complex analysis of the semantic properties of the entire program.

Avoiding storing the value of a variable or temporary that is not subsequently used.

Unreachable or dead code

```
#define DEBUG 0
.....
if (DEBUG)
{.....}
```

8.9.1 Principal Sources of Code Optimizations

The elimination of unreachable code does not usually affect execution speed significantly, but it can substantially reduce the size of the target code.

3) Costly Operations

A code generator should take advantage of opportunities to reduce the cost of operations that are necessary.

The replacement of arithmetic operations by cheaper operations.

For instance, multiplication by 2 can be implemented as a shift operation, and a small integer power, such as x3, can be implemented as a multiplication, such as x*x*x.

This optimization is called reduction in strength.

8.9.1 Principal Sources of Code Optimizations

- Constant folding: use information about constants to remove as many operations as possible or to precompute as many operations as possible.
- Constant propagation: if a variable might also have a constant value for part or all of a program.

8.9.1 Principal Sources of Code Optimizations

Two standard ways to remove procedure calls:

1. Replacing the procedure call with the code for the procedure body (with suitable replacement of parameters by arguments). (Procedure in lining)

2. Recognize tail recursion: when the last operation of a procedure is to call itself. tail recursion removal

8.9.1 Principal Sources of Code Optimizations

```
int gcd ( int u, int v)
    {  if (v == 0)  return u;
       else return  gcd(v, u%v);}    /is a tail recursive

  int fact (int n )
  { if (n==0)  return 1;
   else return n*fact(n-1);  }     / is not a tail recursive
```

8.9.1 Principal Sources of Code Optimizations

```
int gcd ( int u, int v)
    {  if (v == 0)  return u;
       else return  gcd(v, u%v);}
```

```
int gcd( int u, int v)
{ begin:
   if  ( v==0) return u;
   else
  {  int t1 = v, t2 =
     u%v;
      u = t1;  v = t2;
             goto begin;
  }
}
```

8.9.1 Principal Sources of Code Optimizations

4) Prediction Program Behavior:

To perform some of the previously described optimizations, a compiler must collect information about the uses of variables, values and procedures in programs: whether expressions are reused, whether or when variables change their values or remain constant, and whether procedures are called or not.

A different approach is taken by some compilers in that statistical behavior about a program is gathered from actual executions and the used to predict which paths are most likely to be taken, which procedures are most likely to be called often, and which sections of code are likely to be executed the most frequently.

8.9.2 Classification of Optimizations

Two useful classifications :

- The time during the compilation process when an optimization can be applied .
- The area of the program over which the optimization applies.

8.9.2 Classification of Optimizations

1. The time of application during compilation:
Optimizations can be performed at practically every stage of compilation.

For example, constant folding can be performed as early as during parsing. Some optimizations can be delayed until after target code has been generated (*peephole optimizations*).

Typically the majority of optimizations are performed either during intermediate code generation, just after intermediate code generation, or during target code generation.

- source-level optimization
- target-level optimization

8.9.2 Classification of Optimizations

- In ordering the various optimizations, it is important to consider the effect that one optimization may have on another.

```
x = 1;
. . .
y = 0;
. . .
if (y) x = 0;
. . .
if (x) y = 1;
```

```
x = 1;
. . .
y = 0;
. . .
if (0) x = 0;
. . .
if (x) y = 1;
```

```
x = 1;
. . .
y = 0;
. . .
if (x) y = 1;
```

8.9.2 Classification of Optimizations

2. The second classification scheme for optimizations that we consider is by the area of the program over which the optimization

- The categories for this classification are called ***local, global*** and ***interprocedural*** optimizations.

（1）Local optimizations: applied to ***straight-line segments of code***, or ***basic blocks***. That is , code sequences with no jumps into or out of the sequence.

（2）Global optimizations: applied to an individual procedure. Loop discovery are necessary

（3）Interprocedural optimizations: extend beyond the boundaries of procedures to the entire program.

- Interprocedural optimization is even more difficult. .

8.9.3 Data Structures and Implementation Techniques for Optimizations

- Some optimizations can be made by transformations on the syntax tree itself, including constant folding and unreachable code elimination.
- An optimizer construct a graphical representation of the code called a ***flow graph***.
- The nodes of a flow graph are the ***basic blocks***, and the edges are formed from the conditional and unconditional jumps. Each basic block node contains the sequence of intermediate code instructions of the block.

8.9.3 Data Structures and Implementation Techniques for Optimizations

- A flow graph, together with each of its basic blocks, can be constructed by a single pass over the intermediate code. Each new basic block is identified as follows:

(1) The first instruction begins a new basic block;

(2) Each label that is the target of a jump begin a new basic block;

(3) Each instruction that follows a jump begins a new basic block;

8.9.3 Data Structures and Implementation Techniques for Optimizations

- A standard data flow analysis problem is to compute, for each variable, the set of so-called **reaching definitions** of that variable at the beginning of each basic block.
- a **definition** : an intermediate code instruction that can set the value of the variable, such as an assignment or a read.
- A definition is said to **reach** a basic block if at the beginning of the block the variable can still have the value established by this definition.
- Reaching definitions can be used in a number of optimizations.(in constant propagation)

8.9.3 Data Structures and Implementation Techniques for Optimizations

- Another data structure is frequently constructed for each block, called the **DAG of a basic block**.
- DAG traces the computation and reassignment of values and variables in a basic block as follows.

(1) Values that are used in the block that come from elsewhere are represented as leaf nodes.

(2) Operations on those and other values are represented by interior nodes.

(3) Assignment of a new value is represented by attaching the name of the target variable or temporary to the node representing the value assigned.

8.9.3 Data Structures and Implementation Techniques for Optimizations

- Repeated use of the same value also is represented in the DAG structure. For example, the C assignment x = (x+1)*(x+1) translates into the three-address instructions:

$t1 = x + 1$

$t2 = x + 1$

$t3 = t1 * t2$

$x = t3$

8.9.3 Data Structures and Implementation Techniques for Optimizations

- The DAG of a basic block can be constructed by maintaining two dictionaries.

(1) a table containing variable names and constants, with a lookup operation that returns the DAG node to which a variable name is currently assigned.

(2) A table of DAG nodes, with a lookup operation that, given an operation and child node, returns the node with that operation and children or nil if there is no such node.

8.9.3 Data Structures and Implementation Techniques for Optimizations

- Target code, or a revised version of intermediate code, can be generated from a DAG by a traversal according to any of the possible topological sorts of the nonleaf nodes. For example, one of the legal traversal sequences of the three nonleaf nodes of Figure 8.19 would result in the following new sequence of three address instructions, which could replace the original basic block:

  t3 = x - 1

  t2 = fact * x

  x = t3

  t4 = x == 0

  fact = t2

- Of course, wish to avoid the unnecessary use of temporaries, and so would want to generate the following equivalent three-address code, whose order must remain fixed:

  fact = fact * x

  x = x - 1

  t4 = x == 0

8.9.3 Data Structures and Implementation Techniques for Optimizations

- A similar traversal of the DAG of Figure 8.20 results in the following revised three-address code:

  t1 = x + 1

  x = t1 * t1

- Using DAG to generate target code for a basic block, we automatically get local common subexpression elimination.
- The DAG representation also makes it possible to eliminate redundant stores and tells us how many references to each value there are. This gives us information that permits good register allocation.

8.9.3 Data Structures and Implementation Techniques for Optimizations

- A final mothod that is often used to assist register allocation as code generation proceeds involves the maintenance of data called **register descriptors** and **address descriptors.**
- Register descriptors: associate with each register a list of the variable names whose value is currently in that register.
- Address descriptors: associate with each variable name the locations in memory where its value is to be found.

End of Chapter 8