University of California, Irvine

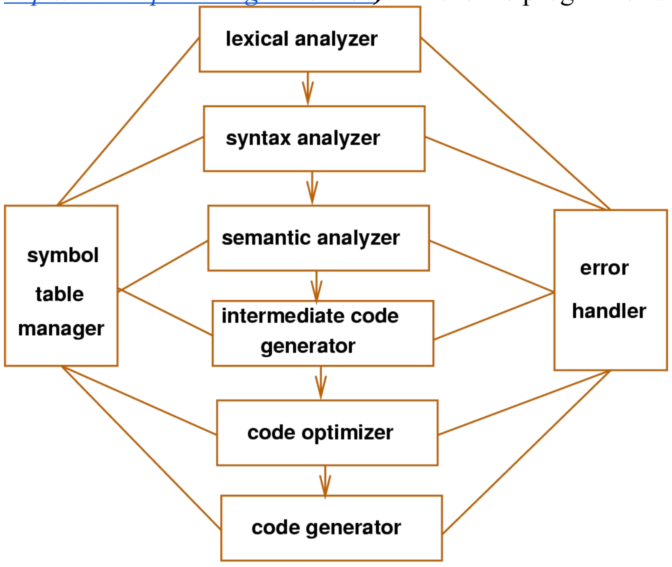
CS243 High performance architecture and compiler Project Report

Name : Jiacheng Feng
Stu ID: 10601635

Table of Contents

1. What Have I Done

I implemented a simple compiler and some optimizations based on PL0 (PL0 grammar: *http://en.wikipedia.org/wiki/PL/0*). This is the program structure.



Instead of using Lexer, I wrote a simple lexical analyzer. After that I implemented a top-down parsing syntax analyzer (Compilers Principles Techniques and Tools, Chapter 4.4 Top-Down Parsing). Designed three-address intermediate code, did some optimization upon intermediate code phase, finally generated 32 bit x86 assembly language.

2. Solved Problems

2.1. Run-time Stack

To design the run-time stack structure, many factors should be taken into account. Meanwhile it must comfort to the feature of x86 machine because the code which it generated runs on it. On the other hand, it must take care of addressing, parameters, return value and return address which is the most important part of run-time stack.

Low Address

Temporary Variables
Local Variables
Pre EBP
Ret Address
ABP(self)
ABP(k)
ABP(k-1)
ABP(k-2)
...
ABP(2)
ABP(1)
ABP(0)
number of parameters
parameter
Ret Value

High Address

When a function call occurs, firstly push back the return value if it has, then the parameter region, then the number of parameters for addressing, then the display region, followed by return address, local variable region, and temporary region.

Here comes my design of display(*http://en.wikipedia.org/wiki/Call_stack, Lexically nested routines*). For each function, in its display region, we do not only record base address of the function who calls it, but also record its own base address at the top. The taking benefit is that we can use the universal way to find base addresses in the display region and then get the offset to address, rather than splitting two conditions to addressing. On the other hand, putting the parameter region at the bottom of stack takes advantage of avoiding waste of memory by not copying parameter repeatedly. However the shortcoming is that we have to push back a value which denotes the number of parameters to addressing the parameter. An example can be seen at appendix A experiment result.

2.2. Three-address Intermediate Code

The design of intermediate code needs to consider both front-end PL0 program and the back-end assembly language. Considering how to do optimization at intermediate code level is needed at the same time. I referred the three-address code in "*Compilers Principles, Techniques and Tools, Chapter 6.2 Three-Address Code*" and did my own intermediate code. Because of no experience before, at the beginning my design is thoughtless which is proved when code the following parts of the compiler. In order to make up problems resulting from the careful less thinking, I have to endlessly add and modify data structure and control flow. Here is my 3-address intermediate code. An example can be seen at appendix A experiment result.

ML_PLUS	0
ML_MINUS	1
ML_TIMES	2
ML_SLASH	3
ML_BEC	4
ML_BEGIN	5
ML_END	6
ML_PUSH	7
ML_POP	8
ML_CALL,	9
ML_CMP	10
ML_JWNT	11
ML_JMP	12
ML_LABEL	13
ML_READ	14
ML_WRITE	15
ML_NULL	16
ML_REG	17

Follows the design details in which each instruction is in such format :TYPE x1,x2,x3.

BEGIN x1,x2,x3 x2.name: function name,x2.adr: the number of local variable;x2.val: the number of parameter;x3.val: the number of temporary variable

Push

Instruction	X1	X2	X3	Information
PUSH		NULL		request for the return space
PUSH	NULL	TYPE_CONST		the number of parameters when function call occurs
PUSH	.val==2			push parameter whose type is var
PUSH	.val==1	Kind>2		push parameter whose type is var and which is array
PUSH	.val==1	Kind<=2		push parameter whose type is var which is not array

Push x2 x1.val record whether the parameter's type is var,1 for yes 2 for no X2.val record whether itself is var

Call x2 x2 function call name

BEC x1,x2,x3 assignment statement x3:=x1
if x3 is array, then x3[x2]=x1
if x1 is array, then x3=x1[x2]

Cmp x1,x2 compare x1 to x2

JWNT x1,x2 if x1 is false then jump to x2.name

Jmp x2 jump to x2.name

LABEL x2 set up lable x2.name

Write x1,x2,x3 x2-output string，x3-output expression

NULL	denote to a null expression
REG x1,x3	store x1's address to register x3

2.3. Common Subexpression Elimination

There is a problem that if we try to do common subexpression elimination after analysis DAG using the method written on the book directly. An example can be seen at appendix B experiment result.Consider the following program.

```
Temp:=x;
x:=y;
y:=x;
```

After doing DAG analysis the x0 node at the beginning could be lost. When an assignment statement refers to x, it will be a new value of x which results to calculation error. To solve this problem, here is my data structure. Firstly I set up two arrays to denote the two tools when doing DAG analysis which are DAG and node table respectively.

```
typedef struct //  nodes in DAG
{
        int         p[MAX_DAG];   //parent node in which p[0] denotes the number
        int         lc,rc;              //left and right child
        TABLE  t;                      //init value
        ML_TYPEc;                   //operator
}NODE1;

typedef struct// table entry of DAG node table
{
        TABLE  t;                      //variable
        int         p;                     //node id
}NODE2;
```

Node1 is the nodes in DAG in which p is an array and p[0] denotes to the number of parent nodes which store in p[]. Lc and Rc denote to the left child and right child respectively. The denoted variable is stored in t. If this node is an intermediate node, then c denotes to the operator of this intermediate node.Node2 is the table entry of DAG node table in which t denotes the corresponding variable and p denotes to the position in DAG.

In order to solve the problem mentioned above resulting from intersecting assignment, when assign each variable to corresponding node, say X, processing the following sentences, if X is assigned in one sentence, it means the value of X changes. Now it will be a error if we use the original node. My solution is that if such situation happens, we refers to a new temporary variable T. Use T instead of X to assign the original node. Thus when X involves in a calculation or assignment next time, T will replace X which solves the problem.

One thing needs to take care is that when we reconstruct intermediate code in order of DAG node, we must process the leaf nodes first. If the variables denoted to leaf nodes change, then we replace it with a temporary variable. When all of the leaf nodes finished this process, then we assign the corresponding variables, and the treat the intermediate node.

Another thing needs to consider is that after DAG optimization, some temporary variables may lost. But there might be some reference at the following sentence such as function call. So my solution is that if such problem occurs, we scan node table again to find the corresponding node. Then the corresponding variable using which to replace the temporary variable to do the function call.

2.4. Other Optimizations

Based on the basic blocks and flow graphs(*Compilers Principles, Techniques and Tools, Chapter 8.4 Basic Blocks and Flow Graphs*), here are two optimizations.An example can be seen at appendix C experiment result.

Peephole optimization(*Compilers Principles, Techniques and Tools, Chapter 8.7 Peephole Optimization*), a simple but effective technique for locally improving the target code which is done by examining a sliding window of target instructions and replacing instruction sequences within the peephole by a shorter or faster sequence.

Constant propagation(*Compilers Principles, Techniques and Tools, Chapter 9.4 Constant Propagation*) replaces expressions that evaluate to the same constant every time they are executed, by that constant. So we just simplify the constant expression at the compile time.

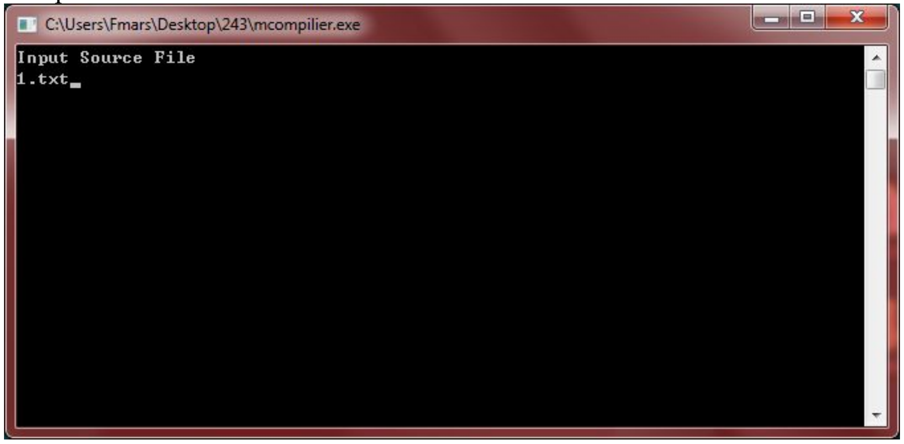
3. What to do next

There are many things need to do and improve. A lot of knowledge needs to know as well. Firstly, it must be about the intermediate code. Due to repetitive modification, the original design has been a mass and hard to read. Thus it must be a better choice to redesign it to be more formative. Secondly, implement a more complex constant propagation take advantage of data-flow values. Thirdly, learn about register relocation which is really an important part of optimization that could improve the speed because there is a significant time cost difference between operation on register and on memory. Finally I want to try to learn some about gcc or llvm which is real world used and cool.

4. How To Run The Program

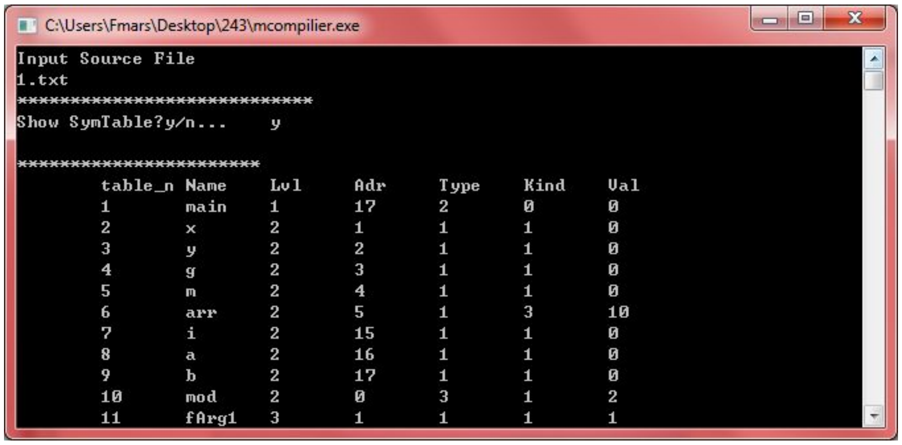
There are three source files header.h, mconpiler.cpp, ml.cpp. After compiling and linking, execute the program, input the source file wanted to be compiled. Then choose whether to display symbol table, intermediate code and whether to do optimizations following the prompting. Finally, use MASM32(*http://www.masm32.com/*) to execute the generated assembly language program.

1. Input the file name



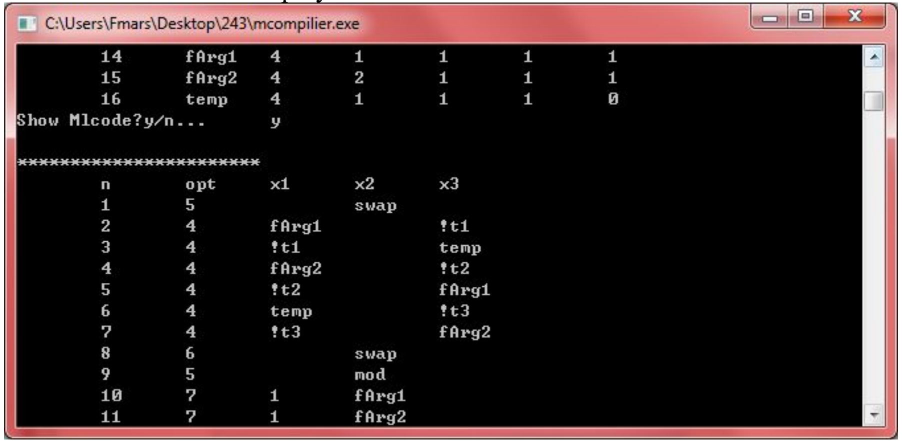
```
Input Source File
1.txt_
```

2. Choose whether to display the symbol table



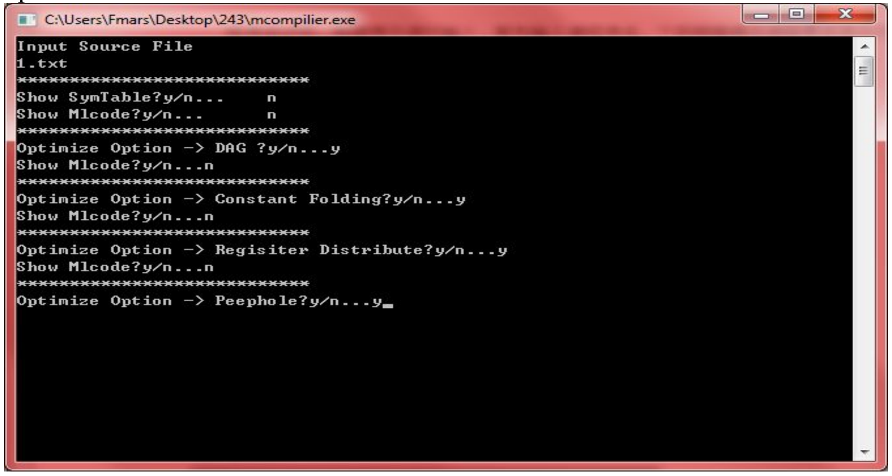
```
C:\Users\Fmars\Desktop\243\mcompilier.exe
Input Source File
1.txt
****************************
Show SymTable?y/n...    y

***********************
        table_n Name    Lvl     Adr     Type    Kind    Val
        1       main    1       17      2       0       0
        2       x       2       1       1       1       0
        3       y       2       2       1       1       0
        4       g       2       3       1       1       0
        5       m       2       4       1       1       0
        6       arr     2       5       1       3       10
        7       i       2       15      1       1       0
        8       a       2       16      1       1       0
        9       b       2       17      1       1       0
        10      mod     2       0       3       1       2
        11      fArg1   3       1       1       1       1
```

3. Choose whether to display the intermediate code



```
C:\Users\Fmars\Desktop\243\mcompilier.exe
        14      fArg1   4       1       1       1       1
        15      fArg2   4       2       1       1       1
        16      temp    4       1       1       1       0
Show Mlcode?y/n...      y

***********************
        n       opt     x1      x2      x3
        1       5               swap
        2       4       fArg1           !t1
        3       4       !t1             temp
        4       4       fArg2           !t2
        5       4       !t2             fArg1
        6       4       temp            !t3
        7       4       !t3             fArg2
        8       6               swap
        9       5               mod
        10      7       1       fArg1
        11      7       1       fArg2
```

4. Choose whether to do optimizations, common subexpression elimination, constant folding, peephole optimization



```
C:\Users\Fmars\Desktop\243\mcompilier.exe
Input Source File
1.txt
****************************
Show SymTable?y/n...    n
Show Mlcode?y/n...      n
****************************
Optimize Option -> DAG ?y/n...y
Show Mlcode?y/n...n
****************************
Optimize Option -> Constant Folding?y/n...y
Show Mlcode?y/n...n
****************************
Optimize Option -> Regisiter Distribute?y/n...y
Show Mlcode?y/n...n
****************************
Optimize Option -> Peephole?y/n...y_
```

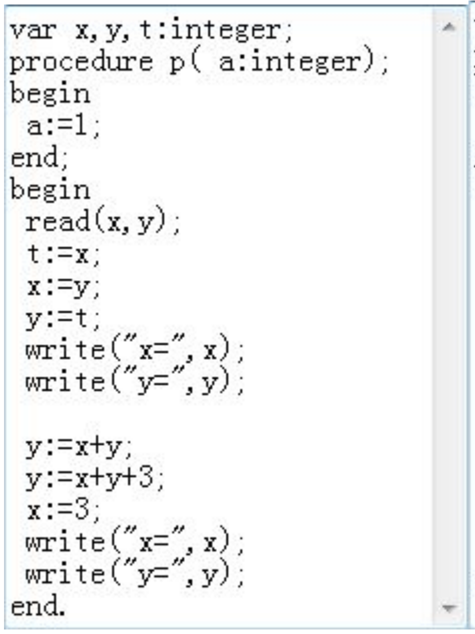
5. Appendix

5.1. Experiment Result A Correctness

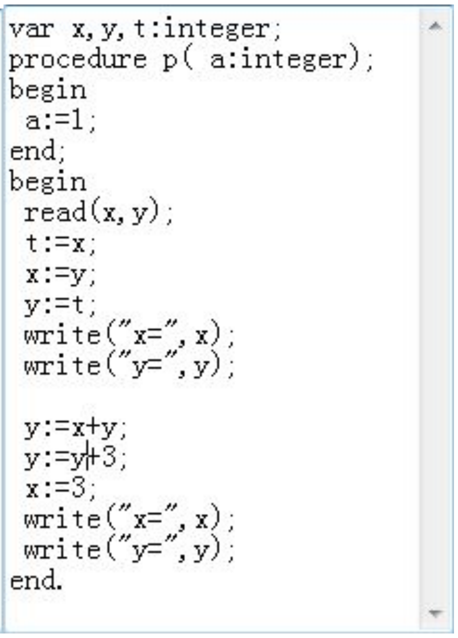
To prove the correctness of compiler design including run-time stack and three-address intermediate code, I designed some source code to compile using the finished compiler. And here are two of the source codes. One is a loop program to prove that the compiler can correctly handle the variables and loop statement and loop nesting. The other is a recursion program to prove that compiler can correctly handle the function call and function return value and recursive function call. It will prove the correctness of run-time stack as well. The source code and execution results can be found at folder /Exp Result/Correctness.

5.2. Experiment Result B CSE

To prove that the common subexpression elimination optimization does work, I designed some experiment to test it and here are two of them.

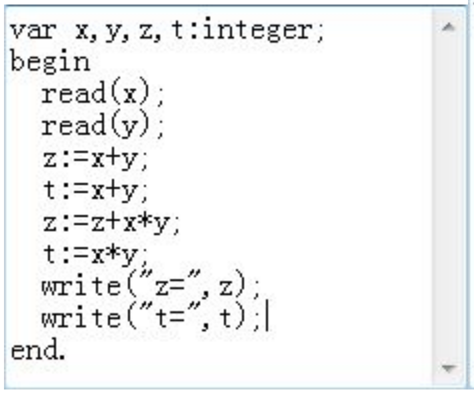


```
var x,y,t:integer;
procedure p( a:integer);
begin
 a:=1;
end;
begin
 read(x,y);
 t:=x;
 x:=y;
 y:=t;
 write("x=",x);
 write("y=",y);

 y:=x+y;
 y:=x+y+3;
 x:=3;
 write("x=",x);
 write("y=",y);
end.
```

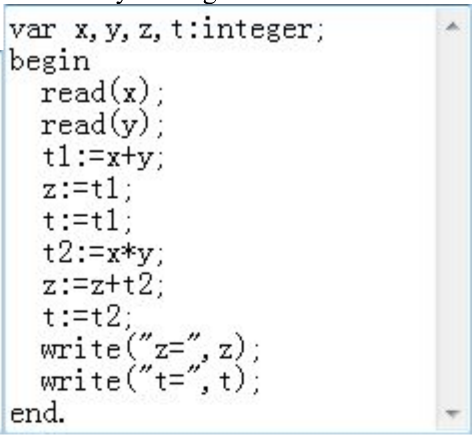


```
var x,y,t:integer;
procedure p( a:integer);
begin
 a:=1;
end;
begin
 read(x,y);
 t:=x;
 x:=y;
 y:=t;
 write("x=",x);
 write("y=",y);

 y:=x+y;
 y:=y+3;
 x:=3;
 write("x=",x);
 write("y=",y);
end.
```

The left is original code and after the optimization it will run in a way like right one.



```
var x,y,z,t:integer;
begin
  read(x);
  read(y);
  z:=x+y;
  t:=x+y;
  z:=z+x*y;
  t:=x*y;
  write("z=",z);
  write("t=",t);
end.
```

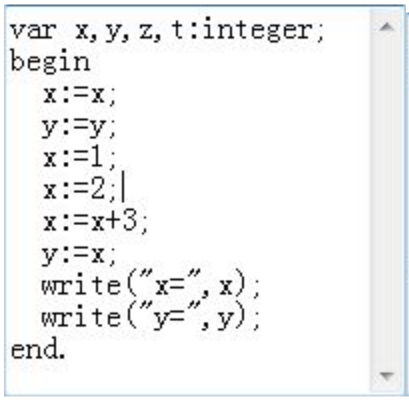


```
var x,y,z,t:integer;
begin
  read(x);
  read(y);
  t1:=x+y;
  z:=t1;
  t:=t1;
  t2:=x*y;
  z:=z+t2;
  t:=t2;
  write("z=",z);
  write("t=",t);
end.
```

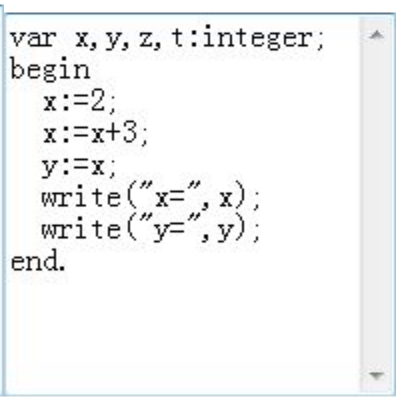
The left is original code and after the optimization it will run in a way like right one which has eliminated the common subexpression x+y and x*y using the temporary variable t1 and t2. The source code and execution results can be found at folder /Exp Result/Common Subexpression Elimination.

5.3. Experiment Result C Others

Here is an simple example shows that my peephole optimization works.



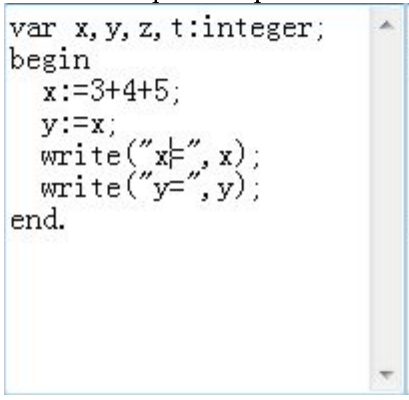
```
var  x,y,z,t:integer;
begin
   x:=x;
   y:=y;
   x:=1;
   x:=2;
   x:=x+3;
   y:=x;
   write("x=",x);
   write("y=",y);
end.
```



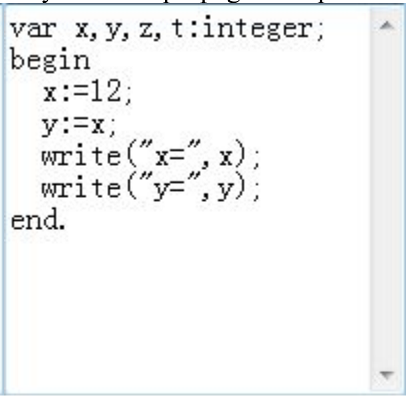
```
var  x,y,z,t:integer;
begin
   x:=2;
   x:=x+3;
   y:=x;
   write("x=",x);
   write("y=",y);
end.
```

The left is original code and after the optimization it will run in a way like right one which eliminates the useless statements such as x:=x , y:=y. The source code and execution results can be found at folder /Exp Result/Others.

Here is an simple example shows that my constant propagation optimization works.



```
var  x,y,z,t:integer;
begin
   x:=3+4+5;
   y:=x;
   write("x=",x);
   write("y=",y);
end.
```



```
var  x,y,z,t:integer;
begin
   x:=12;
   y:=x;
   write("x=",x);
   write("y=",y);
end.
```

The left is original code and after the optimization it will run in a way like right one which will calculate the constant expression at compile time and replace the expression with the value to the target code. The source code and execution results can be found at folder /Exp Result/Others.