CS3300: Compiler Design | Odd Sem

Assignment #4

Optimizations | Deadline: 05/11/2020, 11:55PM

Statement

Write a C/C++ program using lex/yacc that compiles an input C source to an optimized assembly code. The set of optimizations that needs to be performed is mentioned in the following subsections. The optimizations can be done in multiple passes (forward or backward) and can be performed directly at the source level or at the level of AST or IR derived from the source.

In the output, you have to generate two files. The first file is the *assembly file that contains the optimized assembly code*. The second file is the summary file that *contains the summary of all the optimizations that you have performed*. The exact form of the summary file is defined later.

1. Simplification of if-else Blocks

Simplify if-else blocks only when the if-condition contains a literal. For example:

```
20      if (34) {
21              a = c * d;
22      } else {
23              a = e * d;
24      }
```

after elimination of the unreachable block at the source level, it becomes:

```
        a = c * d;
```

The unreachable block does not appear in the generated assembly.

2. Elimination of Unused Variables

Eliminate unused variables. Unused variables are those variables that do not occur in any statement except its own declaration. For example,

```
27      void main() {
28              int x;
29              int y;
30              x = 84;
31                  printf("%d\n", x);
32      }
```

Here, y is an unused variable

3. Strength Reduction of Multiplication

Wherever possible, replace multiplications with a power of 2 by left shift operations.

```
30      x = y * 16;
```

leads to the following assembly:

```
sall    $4, %eax
```

where %eax holds the value of y.

For this optimization, you may assume that in case of multiplication, the operands would never be greater than 1024, and that an overflow would never occur. Also note that if both the operands of a multiplication operation are constant, then instead of this optimization, you have to perform constant folding.

4. Constant Folding

Statically evaluate constant expressions or subexpressions. For example:

```
40      x = (y + 3) * (2 + 4);
```

After constant folding, the expression simplifies to:

```
x = (y + 3) * 6;
```

which is used to generate the assembly.

For this optimization, the testcases will contain only +, - and * operators. Note that you do not need to apply algebraic identities here.

5. Common Subexpression Elimination(CSE)

Eliminate recurring common subexpressions. For example:

```
30      x = a * (b + c);
31      y = d + (b + c);
```

optimizes to, supposing it is done at the IR level:

```
int t;
t = b + c;
x = a * t;
y = d + t;
```

Note that you do not need to apply algebraic identities here.

6. Constant Propagation

Substitute a variable in an expression with its value, if it is statically computable. This applies to the argument of printf statements too. For example,

```
50      x = 3;
51      y = x * 3;
52      z = y * x;
53      printf("%d\n", z);
```

generates the following assembly:

```
#  stores 3, 9, 27 in x, y, z respectively
movl$3, -4(%rbp)
movl$9, -8(%rbp)
movl$27, -12(%rbp)
```

```
# call to printf
movl$27, %esi
leaq.LC0(%rip), %rdi
movl$0, %eax
callprintf@PLT
```

The Summary File Format

The summary file contains one block for each optimization. All line numbers mentioned in the summary file are w.r.t. the original source.

```
unused-vars
[Print all the unused vars, one per line, in the order of their declaration]

if-simpl
[If if-simpl applies, print "0" if the if-block is removed, otherwise print "1".]

strength-reduction
[For every line in the source where strength reduction applies, print:
 L p
where L is the source line number and p is the largest constant operand among all the
generated left-shift operations at line L. The lines in this block must be in increasing order of
L.]

constant-folding
[For every line in the source where constant folding applies, print:
 L c
where L is the source line number and c is the largest value among all the maximal
constant subexpressions at line L. A constant subexpression is maximal if it is not a
part of another, strictly larger, constant subexpression. The lines in this block must be
in increasing order of L.]

constant-prop
[For every line where constant propagation happened, print:
L var1 val1 var2 val2 ... varn valn
where L is the source line number and var1...varn are the variables in the original source that
were replaced with constant values val1...valn respectively, by constant propagation. The lines
must occur in the increasing order of L and the variable names in a line must occur in the order
of their declaration.]

cse
[For every instance of CSE, print:
L1 L2 L3...Ln
where L1 L2 ... Ln are the line numbers of the statements that contain the common
subexpression. The lines in this block must be in increasing order of L1.]
```

Note that the optimizations done on a dead code must not be printed. Dead code includes the unreachable block of an if-else statement including the if-condition, if if-simpl applies.

Order of Optimizations

When more than one kind of optimizations are applicable, there could be cases where one optimization leads to or obviates another optimization. For example, if you have:

```
if (x) {
        y = 3 + 5;
}
```

If x statically evaluates to false (through constant propagation), then if-simpl obviates the need to perform constant folding on y = 3 + 5. You have to handle such cases too.

You are free to choose the order in which you perform the optimizations with the only restriction that CSE must be done only after all the other optimizations have been performed.

Note: A successful submission for Assignment 3 will be provided to you on Moodle. You may use it as a part of your program. Note that you will need to add support of scanf statements in it yourself. You might also have to create new attributes in the AST nodes to hold the relevant source line number(s).

Input

A valid C source code derivable from the grammar described in the "Source Grammar" section. Additionally, we will have the following restrictions:

1. The testcases would not contain loop constructs or nested if. Only if, if-else, assignments, declarations, printf and scanf statements would be present.
2. There will be a maximum of one if/ if-else statement.
3. if / if-else would appear only at the end of main in the testcases. Supporting assignments after if-else would give you extra credit.

You may assume that the C source will have no syntax or semantic errors.

Output

Two files:

1. summary.txt: Contains the summary of the optimizations. The empty blocks also have to be printed.
2. assembly.s: Contains the optimized assembly code

Note: You do not need to print anything on stdout

Examples

Example 1 - Only one kind of optimization

Input

```
1  void main() {
2          int x;
3          int y;
4          scanf("%d", &x);
5  y = (5*3) + x;
6  printf("%d\n", y);
7  }
```

Output - Summary File

unused-vars

if-simpl

strength-reduction

constant-folding
5 15 constant-

prop

cse

Output - Assembly Code

```
.LC0:
        .string         "%d"
.LC1:
        .string         "%d\n"
        .text
        .globl          main
        .type           main, @function
main:
.LFB0:
        .cfi_startproc
        pushq           %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq            %rsp, %rbp
        .cfi_def_cfa_register   6
        subq            $16, %rsp
        leaq            -8(%rbp),   %rax
        movq            %rax, %rsi
        movl            $.LC0, %edi
        movl            $0, %eax
        call            __isoc99_scanf
        movl            -8(%rbp),   %eax
        addl            $15, %eax
        movl            %eax, -4(%rbp)
        movl            -4(%rbp),   %eax
        movl            %eax, %esi
        movl            $.LC1, %edi
        movl            $0, %eax
        call            printf
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
```

Example 2 - Optimizations of Multiple Kinds

Input

```
1  void main() {
2      int a;
```

```
3       int x;
4       int y;
5       int z;
6       z = 3;
7       x = z + 4;
8       y = x + 8;
9       printf("%d\n", x);
10      scanf("%d", &x);
11
12      if (y == 0) {
13          z = (2 * 3) + 14;
14          printf("%d\n", z);
15      } else {
16          z = x * 64;
17          y = x + y;
18          printf("%d\n", y);
19          printf("%d\n", z);
20          y = (x*3) + 5;
21          z = (x*3) + 7;
22          printf("%d\n", y);
23          printf("%d\n", z);
24      }
25  }
```

Output - Summary File

```
unused-vars
a

if-simpl
0

strength-reduction
16 6

constant-folding
7 7
8 15

constant-prop
7 z 3
8 x 7
9 x 7
17 y 15

cse
20 21
```

Explanation:

- unused-vars: a is the only unused variable
- if-simpl: The if-condition evaluates to false, and hence the if-block is removed line 11.
- strength-reduction: multiplication by 64 becomes left shift by 6

- constant-folding: The constant expressions at line 7 and 8 are evaluated to produce 7 and 15 respec-tively.
- constant-prop: The variables occuring in the specified lines are replaced with their values
- CSE: Line 20 and 21 have the common subexpression (x*3)

Output - Assembly Code

```
.LC0:
        .string         "%d\n"
.LC1:
        .string         "%d"
        .text
        .globl          main
        .type           main, @function
main:
.LFB0:
        .cfi_startproc
        pushq           %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq            %rsp, %rbp
        .cfi_def_cfa_register 6
        subq            $16, %rsp
        movl            $3, -4(%rbp)
        movl            $7, -16(%rbp)
        movl            $15, -8(%rbp)
        movl            $7, %esi
        movl            $.LC0, %edi
        movl            $0, %eax
        call            printf
        leaq            -16(%rbp), %rax
        movq            %rax, %rsi
        movl            $.LC1, %edi
        movl            $0, %eax
        call            __isoc99_scanf
        movl            -16(%rbp), %eax
        sall            $6, %eax
        movl            %eax, -4(%rbp)
        movl            -16(%rbp), %eax
        addl            $15, %eax
        movl            %eax, -8(%rbp)
        movl            -8(%rbp), %eax
        movl            %eax, %esi
        movl            $.LC0, %edi
        movl            $0, %eax
        call            printf
        movl            -4(%rbp), %eax
        movl            %eax, %esi
        movl            $.LC0, %edi
        movl            $0, %eax
        call            printf
        movl            -16(%rbp), %edx
        movl            %edx, %eax
        addl            %eax, %eax
```

```
addl    %edx, %eax
movl    %eax, -12(%rbp)
movl    -12(%rbp), %eax
addl    $5, %eax
movl    %eax, -8(%rbp)
movl    -12(%rbp), %eax
addl    $7, %eax
movl    %eax, -4(%rbp)
movl    -8(%rbp), %eax
movl    %eax, %esi
movl    $.LC0, %edi
movl    $0, %eax
call    printf
movl    -4(%rbp), %eax
movl    %eax, %esi
movl    $.LC0, %edi
movl    $0, %eax
call    printf
leave
.cfi_def_cfa 7, 8
ret
.cfi_endproc
```

Making a Submission

1. Create a tar.gz file with filename as <ROLLNO>.tar.gz (eg. CS12B043.tar.gz) having the following structure:
   - CS18B043 <directory>
     – *.l
     – *.y
     – Makefile

   The Makefile should run lex, yacc, compile the generated code and generate an executable a.out file.
2. To make sure the submission runs properly, evaluate it locally using the evaluation script provided on Moodle. To get help with the evaluation script, run ./eval.py -h
3. Upload it on Moodle.

Evaluation of a Submission

There are a total of 44 test cases - 8 of them are public and the rest are private. For every test case, we will perform the following checks:

1. The executable from your assembly will be differentially tested against the executable generated from the original source by gcc.
2. The assembly code generated by your program will be checked to make sure that it is optimized.
3. Your optimization summary file will also be checked. Make sure it lists all the optimizations properly.

Testcases Description

The following table describes the testcases:

Optimization Type(s)	No of Test Cases	Total Marks
if-simpl basic	4	2
unused-var	4	2
strength-reduction	4	2
constant-folding	4	2
common-subexpr	4	2
const-prop	4	2
Multiple types excluding if-simpl advanced	16	8
Multiple types including if-simpl advanced*	4	2(bonus)

* if-simpl advanced refers to the case where assignment statements after if-else are supported

Source Grammar

The productions in green and red are additions and deletions, respectively, w.r.t. the grammar in assignment 3.

program ! decl_list

decl_list ! decl_list decl j decl

decl ! var_decl j func_decl

var_decl ! type_spec identifier ";"

j type_spec identifier "," var_decl

j type_spec identifier "[" integerLit "]" ";"

j type_spec identifier "[" integerLit "]" "," var_decl

type_spec ! "void" | "int" | "float"

j "void" "*" | "int" "*" | "float" "*"

fun_decl ! type_spec identifier "(" params ")" compound_stmt

params ! param_list |

param_list ! param_list "," param | param

param ! type_spec identifier | type_spec identifier "[" "]"

stmt_list ! stmt_list stmt | stmt

stmt ! assign_stmt | compound_stmt | if_stmt | while_stmt | print_stmt j

return_stmt | break_stmt | continue_stmt | scan_stmt

expr_stmt ! expr ";"

while_stmt ! "while" "(" expr ")" stmt

print_stmt ! "printf("format_specifier"," identifier");"

scan_stmt ! "scanf(“%d\” , &" identifier ");"

compound_stmt ! "{" local_decls stmt_list "}"

local_decls ! local_decls local_decl |

local_decl ! type_spec identifier ";"

j type_spec identifier "[" expr "]" ";"

if_stmt ! "if" "(" expr ")" stmt

j "if" "(" expr ")" stmt "else" stmt

return_stmt ! "return" ";" | "return" expr ";"

break_stmt ! "break" ";"

continue_stmt ! "continue" ";"

assign_stmt ! identifier "=" expr ";" | identifier "[" expr "]" "=" expr ";"

expr ! Pexpr "||" Pexpr

! Pexpr "==" Pexpr | Pexpr "!=" Pexpr

! Pexpr "<=" Pexpr | Pexpr "<" Pexpr | Pexpr ">=" Pexpr | Pexpr ">" Pexpr

! Pexpr "&&" Pexpr

! Pexpr "+" Pexpr | Pexpr "-" Pexpr

! Pexpr "*" Pexpr | Pexpr "/" Pexpr | Pexpr "%" Pexpr

! "!" Pexpr | "-" Pexpr | "+" Pexpr | "*" Pexp | "&" Pexp

! Pexpr

! identifier "(" args ")"

! identifier "[" expr "]"

Pexpr ! integerLit | floatLit | identifier | "(" expr ")"

integerLit ! <INTEGER_LITERAL>

floatLit ! <FLOAT_LITERAL>

identifier ! <IDENTIFIER>

format_specifier ! ""“%d\n""

arg_list ! arg_list "," expr | expr

args ! arg_list |