COMP3109 Assignment 3

Writing a Compiler – Group Assignment (10 marks)

This third assignment is a **group assignment**, and is due **in Week 11**. Your assignment will not be assessed unless all two of the following criteria are met:

1. Hand in a signed group academic honesty form in the tutorial
2. Submit a tarball of your source code to WebCT for all solved tasks with plenty of remarks, i.e., documentation should be in form of remarks.

Please form groups of two or three students (groups of three preferred).

Task 1 (10 marks)

We want to implement a mini compiler for a vector language in a compiler language tool of your choice that is able to deal with attribute grammars (e.g. ANTLR). A program in the vector language consists of several functions. Each function has several input and output vectors. We further assume that all vectors of a function have the same length. A function in the vector language consists of a sequence of vector assignments or calls to other functions defined in the vector language. The grammar of the language is given below, where M is the start symbol.

$$
\begin{array}{rcl}
M & \rightarrow & \varepsilon \\
M & \rightarrow & M\ F \\
F & \rightarrow & \textbf{func ident}\ P\ D\ S\ \textbf{end} \\
P & \rightarrow & \varepsilon \\
P & \rightarrow & \textbf{(}\ L\ \textbf{)} \\
L & \rightarrow & \textbf{ident} \\
L & \rightarrow & L\ \textbf{, ident} \\
D & \rightarrow & \textbf{var}\ L\ \textbf{;} \\
D & \rightarrow & \varepsilon \\
S & \rightarrow & \mathrm{S}\ \textbf{;}\ \mathrm{S} \\
S & \rightarrow & \textbf{ident =}\ E
\end{array}
\qquad
\begin{array}{rcl}
S & \rightarrow & \textbf{ident (}\ L\ \textbf{)} \\
S & \rightarrow & \varepsilon \\
E & \rightarrow & E\ \textbf{-}\ E \\
E & \rightarrow & E\ \textbf{+}\ E \\
E & \rightarrow & E\ \textbf{*}\ E \\
E & \rightarrow & E\ \textbf{/}\ E \\
E & \rightarrow & \textbf{min (}\ E\ \textbf{,}\ E\ \textbf{)} \\
E & \rightarrow & \textbf{(}\ E\ \textbf{)} \\
E & \rightarrow & \textbf{ident} \\
E & \rightarrow & \textbf{num}
\end{array}
$$

Write an attribute grammar that translates an input program in the vector language to an assembly file. The assembly output should take advantage of the SSE extension (high speed floating point operations which can operate on 4 single precision numbers at the same time) of the IA32 architecture. The produced assembly code should define functions that can be used in C programs. The structure of this exercise is to put together assembly templates (i.e. blocks of text) which are given in the rest of this assignment. These text blocks need to be pasted and written to a file. Some of these text blocks need specific parameters that are to be computed by the attributes of the grammar.

For example the following program in the vector language defines a function **mymin**

```
func mymin(a, b, c)
 c = min(a,b) + 20
end
```

that has three parameters **a**, **b** and **c**. The element-wise minimum is computed, the constant vector 20 is added, and the result is stored in parameter **c**. The function **mymin** can be used as sketched in the following C code fragment:

```c
#include <stdlib.h>

/* alignment macro: aligns a memory block a to multiplies of a */
#define align(s,a) ((size_t)((s) + ((a) - 1)) & ~((size_t) (a) - 1))
/* Alignment for SSE unit */
#define SSE_ALIGN (16)
/* Number of elements */
#define NUM (100)

extern void mymin(int, float *, float *, float *);

int main(void) {
  float *a = malloc(sizeof(float)*NUM + SSE_ALIGN),
        *b = malloc(sizeof(float)*NUM + SSE_ALIGN),
        *c = malloc(sizeof(float)*NUM + SSE_ALIGN);
  /* make sure that pointers are aligned to multiplies of 16 bytes */
  a = (float *) align(a, SSE_ALIGN);
  b = (float *) align(b, SSE_ALIGN);
  c = (float *) align(c, SSE_ALIGN);
  ...
  /* write values to a and b */
  ...
  /* invoke the function written in the vector language */
  mymin(NUM, a, b, c);
  ...
  /* read values from c */
  ...
  return 0;
}
```

In this code fragment three single precision floating point arrays **a**, **b** and **c** are declared and used as parameters of the vector language function **mymin**. Before using the arrays as vectors they need to be aligned to multiplies of 16 – this is an SSE requirement. Additionally, we assume that the length of a vector is a multiple of 4 because a single SSE instruction can operate on 4 single precision floating point numbers at the same time. The first parameter gives the length of the vectors followed by the vector parameters. Depending on the semantics of the function, the function either reads or writes from the arrays.

When you write assembly code you need to be aware of how the program stack is structured (see textbook) so that you can access the length of the vectors and vector parameters. It is important to remember that the program stack grows downwards.

...	higher addresses	
second vector parameter	16	*third parameter*
first vector parameter	12	*second parameter*
length of vectors	8	*first parameter*
return address	4	*saved by the call instructions*
frame pointer of calling functions	0	*saved as soon as the function is invoked*
first free element on stack	-4	*value of frame pointer*
...	lower addresses	

The stack pointer in IA32 is denoted by `%esp` and the frame pointer by `%ebp`. A function definition in the vector language uses following template to produce assembly code:

```
.text
.align 4,0x90

.global <name>
<name>:
   pushl    %ebp
   movl     %esp, %ebp
   pushl    %ebx          # save %ebx

   #  <Insert the body of function here>

   movl     -4(%ebp), %ebx
   leave
   ret
```

where `.text` denotes that the machine code should be stored in the text segment (see textbook) of the memory. The next pseudo-mnemonic makes sure that the address of the function is aligned to multiples of fours. The pseudo-mnemonic `.global <name>` defines a global name `<name>` for the linker. With `<name>:` you define an address for the function entry point. The next instruction pushes the frame pointer of the calling function onto the stack. The second instruction moves the current value of the stack pointer to the frame pointer. The third instruction saves register `%ebx` onto to stack since it is callee save, i.e., the callee expects that `%ebx` does not change its value.

After the first three instructions you add the code of reserving space on the stack for local variables and the assignments of the vector function. A function definition ends with loading the previous value of ebx from the stack **`leave`**, which unloads the frame pointer. Instruction **`ret`** loads the return address into the instruction pointer of the CPU and returns to the next instruction.

Local variables in the vector language are stored on the stack from `-16(%ebp)` onward. The memory block for all local variables are created by subtracting the product of the length of a vector and the number of local stack variables from the stack pointer and adding a padding for pointer alignment, i.e.,

```
movl     8(%ebp), %eax          # allocate local variables
imull    $4*<N>, %eax, %eax
addl     $15, %eax
subl     %eax, %esp
```

where `<N>` is the number of local variables. The number of local variables can be computed by an attribute in your attribute grammar. This text block needs to be generated as a first text block in the function body.

The following three templates are concerned about computing addresses of vector parameters, local variables, and constants. These three text blocks will be used for address computations in templates that either compute new vectors or assign a vector/constant to another vector. The address of a vector parameter is computed by

```
movl    8+4*<N> (%ebp), <destreg>
```

where `<N>` is the nth parameter of the function (counting from 1). Basically, these are pointers which are pushed by the C function onto the stack. For the following templates the destination register could be either `%eax`, `%ebx`, or `%edx`. The address of a local variable is computed by

```
movl     8(%ebp), <destreg>
imull    $4*<N>, <destreg>, <destreg>
addl     $16, <destreg>
subl     %ebp, <destreg>
negl     <destreg>
sub      $15, <destreg>
andl     $-16, <destreg>
```

where `<N>` is the nth local variable (starting from 1) of the vector function. The address has to be computed at runtime because the length of the vectors is not known ahead of time. The last template computes the address of a constant `<X>`:

```
movl     $.const<X>, <destreg>
```

where the label `.const<X>` is a new label. For generating constants we need a postamble at the end of the assembly code that describes the constant:

```
.data
.align 16
.const<X>:
   .float <X>
   .float <X>
   .float <X>
   .float <X>
```

where `<X>` is the floating point number `X`. We need four repetitions because an `SSE` operation does four operations in a single step.

In the following we introduce assembly code templates for basic forms. The first form we outline the templates for are assignment statements. These are statements of the form

ident = factor

or

ident = factor **op** factor

where "factor" is either a variable or a constant. Conceptually, you break up the right-hand side of an expression in basic forms. In this conceptual transformation you will introduce numerous local variables, e.g., the input statement

```
x = a + b + c + d
```

will be conceptually transformed to

```
x1 = a + b;
x2 = x1 + c;
x3 = x2 + d;
```

Note that I do not suggest to perform this transformation. I suggest to create local variables on the fly by using attributes and an expression node in the syntax tree representing a basic form. Furthermore, you want to reuse local variables as much as you can too avoid stack size explosion.

The basic form "**ident** = factor" is translated to assembly with following template:

```
    <load source address into  %eax>
    <load destination address into %edx>

    movl   8(%ebp), %ecx
    shrl   $2, %ecx
    jz     .loop_end<X>

.loop_begin<X>:
    movaps (%eax), %xmm0
    movaps %xmm0, (%edx)

    addl   $16, %eax #add this line if %eax is not pointing to a constant
    addl   $16, %edx
    loopl  .loop_begin<X>
.loop_end<X>:
```

where <X> is a unique number for this basic form. This text block loads the source and destination address into eax and edx by using the templates as previously shown. The assembly code inside loads the vector length and divides it by 4 (i.e. shift of number by two digits). Inside the loop the moveaps instructions load the instructions into the first SSE register. After performing the move instruction, the first SSE register will contain 4 consecutive floating point numbers. With the second moveaps instruction the 4 consecutive floating point numbers are written to the destination.

The basic form **ident** = factor **op** factor is translated with following template:

```
    <load source1 address into  %eax>
    <load source2 address into  %ebx>
    <load destination address into %edx>

    movl   8(%ebp), %ecx
    shrl   $2, %ecx
    jz     .loop_end<X>

.loop_begin<X>:
    movaps (%eax), %xmm0
    movaps (%ebx), %xmm1
    <operation> %xmm0, %xmm1
    movaps %xmm1, (%edx)

    addl   $16, %eax #add this line if %eax is not pointing to a constant
    addl   $16, %ebx #add this line if %ebx is not pointing to a constant
    addl   $16, %edx
    loopl  .loop_begin<X>
.loop_end<X>:
```

Use the operation **addps** for addition, **subps** for subtracting, **divps** for division, **mulps** for multiplication, and **minps** for minimum.

The second statement is that of a function call, which are of the form

ident(arg1, ..., argN)

In order to invoke a function, three things need to happen. First, the arguments to the function need to be pushed onto the program stack. Once these have been pushed, the function can then be invoked. Because the program stack is in fact a stack, the arguments `arg1, ..., argN` need to be pushed onto the stack in reverse order. Once the function has returned from invocation, the stack needs to be restored to the size that it was before pushing on the arguments. Remember that all function calls will need the implicit "length of vectors" value as the first argument to the function. So invoking a function of arity n requires placing $n + 1$ items on the stack.

```
    # invoke function <name>
    subl   $4*<N+1>, %esp    # grow the stack
    movl   <argN>, %eax      # setup argN
    movl   %eax, <4*N>(%esp)
    ...
    movl   <arg1>, %eax      # setup arg1
    movl   %eax, 4(%esp)
    movl   8(%ebp), %eax     # setup implicit vector length
    movl   %eax, 0(%esp)
    call   <name>            # invoke the function
    addl   $4*<N+1>, %esp    # restore the stack
```

After you have written the stub in C and produced code you can create an executable with

```
gcc -Wall -W -g -o myexe stub.c compiled.s
```

and you can trace the executable with `gdb`. The emission of the assembly code might be done as a traversal in your parser generator tool to avoid awkward data structures for storing the text file.

For speed fanatics: you can achieve substantial more speed by unrolling the loop and pre-computing the addresses of local variables.

Two scripts which might come in use. This first script called `compile.sh`, is used to compile our sample solution to the final executable. Our ANTLR grammar file is called `VPL.g` (Vector Processing Language).

```bash
#!/bin/bash
set -e

BUILD_DIR=build

if [ ${#} != 1 ]; then
  echo "Usage: ${0} filename.vpl" >&2
  exit 1
fi

# builds the ANTLR-generated parser using the grammar file
java -cp antlr-3.1.2.jar org.antlr.Tool -o ${BUILD_DIR} VPL.g
touch ${BUILD_DIR}/__init__.py

# uses the ANTLR-generated parser to convert the VPL program to ASM
./vpl2asm.py < ${1} > ${1}.s

# compiles the ASM and C file together
gcc -Wall -W main.c ${1}.s -o my_program
```

The second script, `vpl2asm.py`, invokes our Python parser generated by ANTLR:

```python
#!/usr/bin/env python
import sys
import antlr3
from build.VPLLexer import VPLLexer
from build.VPLParser import VPLParser

char_stream = antlr3.ANTLRInputStream(sys.stdin)
lexer = VPLLexer(char_stream)
tokens = antlr3.CommonTokenStream(lexer)
parser = VPLParser(tokens)
root = parser.prog()
```

These scripts are useful if you use Python as your target langauge within ANTLR. However, they can be easily adapted for your own chosen target language.

Task 2 (0 marks)

Convert the input program to a LISP/SCHEME list and write a simplification function that simplifies the input. Consider the following:

- apply simple algebraic laws to simplify the expression, e.g., **x*0**, **1+x+1**, etc.
- remove unnecessary assignments that do not contribute to the output.
- identify common sub-expression and replace them by a local variable so that common sub-expressions need to be computed online once. Make sure that the semantics of the input program is not destroyed (e.g., more than one assignment of a variable, etc.)