Computer Science & Engineering Department
I. I. T. Kharagpur

Compilers Laboratory: CS39003
3rd Year CSE: 5th Semester

Assignment – 5: Target Code Generator for ***miniMatlab*** *Marks: 100*
Assign Date: *11^{th} October, 2017* Submit Date: *23:55, 3^{rd} November, 2017*

1 Preamble – miniMatlab

The Lexical Grammar and the Phase Structure Grammar (Assignment 3) for **miniMatlab** have already been defined in Assignment 3 using the International Standard **ISO/IEC 9899:1999 (E)**. Finally, the 3–Address Code structure and a further subset of **miniMatlab** has been specified (Assignment 4) for translating the input **miniMatlab** program to an array of 3–Address quad's, a supporting symbol table, and other auxiliary data structures.

In this assignment you will write a target code translator from the array of 3–Address quad's (with the supporting symbol table, and other auxiliary data structures) to the assembly language of x86-64. The translation is now machine-specific and your generated assembly code would be translated with the gcc assembler to produce the final executable codes for the **miniMatlab** program.

2 Scope of Target Translation

- For simplicity restrict **miniMatlab** further:
  1. Skip shift and bit operators.
  2. Support only **void**, **int**, **float**, **Matrix** and **char** types. float or double?
  3. Support only 2–dimensional Matrices.
  4. Support only **void**, **int**, **float**, **Matrix**, **char**, **void***, **int***, and **char*** as function return types.
  5. No type conversion will be supported.
- For I/O, provide a library (as created in Assignment 2) using in-line assembly language program of x86-64 along with `syscall` for gcc assembler.:
  - `int printStr(char *)` – prints a string of characters. The parameter is terminated by '`\0`'. The return value is the number of characters printed.
  - `int printInt(int n)` – prints the integer value of `n` (no newline). It returns the number of characters printed.
  - `int readInt(int *n)` – reads an integer (signed) and returns it. The parameter is for error (`ERR = 1, OK = 0`).
  - `int readFlt(float *f)` - reads a floating point number in '`%f`' format e.g. `-123.456`. Caller get the value through the pointer parameter. The return value is `ERR` or `OK`.
  - `int printFlt(float f)` - prints the floating point number passed as parameter. Returns the number of characters printed.
  - Utilize the above functions for reading and printing matrices.

  The header file `myl.h` of the library will be as follows:

```
#ifndef _MYL_H
#define _MYL_H
#define ERR 1
#define OK 0
int printStr(char *);
```

```
int printInt(int);
int readInt(int *n);  // *n is for error, if the input is not an integer
int readFlt(float *f);
int printFlt(float f);
#endif
```

3 Design of the Translator

In this assignment, however, you do not need to deal with any machine-independent or machine-specific optimization. Hence the translation comprises the following major steps only:

Memory Binding This deals with the design of the allocation schema of variables (including parameters and constants) that associates each variable to the respective address expression or register. This needs to handle the following:

- *Handle local variables, parameters, and return value for a function.* These are automatic and reside in the Activation Record (AR) of the function. Various design schema for AR are possible based on the calling sequence protocol. A sample AR design could be as follows:

Offset	Stack Item	Responsibility
–ve	Saved Registers	Callee Saves & Restores
–ve	Callee Local Data	Callee defines and uses
0	Base Pointer of Caller	Callee Saves & Restores
	Return Address	Saved by call, used by ret
+ve	Return Value	Callee writes, Caller reads
+ve	Parameters	Caller writes, Callee reads

Activation Record Structure with Management Protocol

  - Offsets in the AR are with respect to the Base Pointer of Callee.
  - Return Value can alternatively be returned through a register (like accumulator).
  - The AR will be populated from the Symbol Table of the function.
  - Symbol Tables of nested blocks will be flattened and its variables allocated within the Symbol Table (and hence the AR) of the function where there occur in. Necessary name mangling will be performed to to take care of same lexical name for different variables in different nested scopes.

- *Handle global variables* (note that local static variables are not allowed in **miniMatlab**) as static and generate allocations in static area. This will be populated from ST.gbl.
- *Generate Constants from Table of Constants* – handle string constants as assembler symbols in Data Segments and integer constants as parts of target code (Text Segment)
- *Register Allocations & Assignment*: Create memory binding for variables in registers:
  - After a load / store the variable on the activation record and the register have identical values
  - Registers can be used to store temporary computed values
  - Register allocations are often used to pass parameters
  - Register allocations are often used to return values

Note: *Refer to* Run-Time Environment *lecture presentations for details and examples on memory binding.*

Code Translation This deals with the translation of 3–Address quad's to x86-64 assembly code. This needs to handle:

- *Generation of Function Prologue* – few lines of code at the beginning of a function, which prepare the stack and registers for use within the function.
- *Generate Function Epilogue* – appears at the end of the function, and restores the stack and registers to the state they were in before the function was called.
- *Map 3–Address Code to Assembly* – to translate the function body do:

- Choose optimized assembly instructions for every expression, assignment and control quad.
- Use algebraic simplification & reduction of strength for choice of assembly instructions from a quad.
- Use Machine Idioms (like **inc** for i++ or ++i in place of **add reg, 1**).

Note: *Refer to* Target Code Generation *lecture presentations for details.*

Target Code Integrate all the above code into an Assembly File for gcc assembler.

4 The Assignment

1. Write a target code (x86-64) translator from the 3-Address quad's generated from the flex and Bison specifications of **miniMatlab** (with restrictions as mentioned in Section 2). Assume that the input **miniMatlab** file is lexically, syntactically, and semantically correct. Hence no error handling and / or recovery is expected.

2. Prepare a Makefile to compile and test the project.

3. Prepare test input files ass5_*roll*_test<number>.c to test the target code translation and generate the translation output in ass5_*roll* <number>.asm.

4. Name your files as follows:

File	Naming
Flex Specification	ass5_*roll*.l
Bison Specification	ass5_*roll*.y
Data Structures (Class Definitions) & Global Function Prototypes	ass5_*roll*_translator.h
Data Structures, Function Implementations & 3–Address Translator	ass5_*roll*_translator.cxx
Target Translator & x86-64 Translator main()	ass5_*roll*_target_translator.cxx
Test Inputs	ass5_*roll*_test<number>.c
3–Address Test Outputs	ass5_*roll*_quads<number>.out
Test Outputs	ass5_*roll* <number>.asm

5. Prepare a zip file with the name ass5_*roll*.zip containing all the files and upload to Moodle.

5 Credits

Design of Memory Binding: **15 + 5 + 5 + 5 + 10 = 40**
Handling of Activation Records
Handling of Nested Symbol Tables
Handling of Static Memory & Binding
Handling of Constants
Handling of Register Allocation & Assignment

Design of Code Translation: **5 + 5 + 10 = 20**
Handling of Prologue
Handling of Epilogue
Handling of Function Body

Design of Target Code Management: **10**
Integration of translated codes into an assembly file

Design of Test files and correctness of outputs: **10 + 10 = 20**
Test at least 5 i/p files covering all rules
Shortcoming and / or bugs, if any, should be highlighted

Integrated interface of the **miniMatlab** Compiler: **10**