Computer Science and Engineering Department
Indian Institute of Technology Kharagpur

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

Assignment – 6: Target Code Generator for tinyC — *Marks: 100*
Assign Date: *November 2, 2020* Submit Date: *23:55, November 12, 2020*

1 Preamble – tinyC

The Lexical Grammar (Assignment 3) and the Phase Structure Grammar (Assignment 4) for the language tinyC have already been defined as subsets of the C language specification from the International Standard **ISO/IEC 9899:1999 (E)**. Finally, three address code (TAC) structure and a further subset of tinyC has been specified (Assignment 5) for translating the input tinyC program to TAC quad array, a supporting symbol table, and other auxiliary data structures.

In this assignment you will be required to write a target code translator from the TAC quad array (with the supporting symbol table, and other auxiliary data structures) to the assembly language of x86-32. 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 tinyC program.

2 Scope of Target Translation

- For simplicity restrict tinyC further:
  1. Skip shift and bit operators.
  2. Support only **void**, **int**, and **char** types. Skip **double** type.
  3. Support only one–dimensional arrays.
  4. Support only **void**, **int**, **char**, **void***, **int***, and **char*** for returns types of functions.
  5. No type conversion to be supported.
- For input/output, provide a library (similar to that created in Assignment 2) using in-line assembly language program of x86-32 along with `syscall` for gcc assembler. The library will contain the following functions:
  a) `int printStr(char *)` – prints a string of characters. The parameter is a character array terminated by '`\0`'. The return value is the number of characters printed.
  b) `int printInt(int n)` – prints the integer value of `n` (no newline). It returns the number of characters printed.
  c) `int readInt(int *eP)` – reads an integer (signed) and returns it. The parameter eP is for error reporting ($ERR = 1$ for error condition, and $OK = 0$ for no error).

  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 *eP);  // *eP is for error, if the input is non-integer
#endif
```

3 Design of the Translator

The steps for target code generation have been outlined in Target Code Generation lecture presentations. In this assignment, however, you do not need to deal with any machine-independent or machine-specific optimization. Hence the translation will comprise of the following major steps only:

a) **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 variables and will reside in the Activation Record (AR) of the function. Various design schema for AR are possible based on the calling sequence protocol. A sample Activation Record structure along with the management protocol is shown below:

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

  The following points may be noted:
  - Offsets in the AR are with respect to the Base Pointer of Callee.
  - Return Value can alternatively be returned through a register.
  - The AR will be populated from the Symbol Table of the function.
  - Symbol Tables of nested blocks will be flattened and their variables allocated within the Symbol Table (and hence the AR) of the function where they 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 tinyC) as static and generate allocations in static area. This will be populated from global symbol table (ST.gbl).
- *Generate Constants from Table of Constants* – handle string constants as assembler symbols in DATA SEGMENT and integer constants as parts of target code in TEXT SEGMENT.
- *Register Allocations & Assignment*: Create memory binding for variables in registers with the following considerations:
  - After a load / store the variable on the activation record and the register will have identical values.
  - Registers can be used to store temporary computed values.
  - Register allocations are often used to pass `int` or pointer parameters.
  - Register allocations are often used to return `int` or pointer values.

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

b) **Code Translation**: This deals with the translation of 3-address quad's to x86-32 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 using the following rules:
  - Choose optimized assembly instructions for every expression, assignment and control quad.
  - Use algebraic simplification and reduction of strength for choice of assembly instructions from a quad.
  - Use machine idioms (like **inc** for `i++`, or **add reg, 1** for `++i`).

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

c) **Target Code**: Integrate all the above code into an Assembly File for gcc assembler.

4 The Assignment

1. Write a target code (x86-32) translator from the 3-address quads generated from the flex and bison specifications of tinyC (with restrictions as mentioned in Section 2). Assume that the input tinyC 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 `ass6_`*roll*`_test<number>`.c to test the target code translation and generate the translation output in `ass6_`*roll* `<number>`.asm.

4. Name your files as follows:

File	Naming
Flex Specification	`ass6_`*roll*.l
Bison Specification	`ass6_`*roll*.y
Data Structures (Class Definitions) and Global Function Prototypes	`ass6_`*roll*_translator.h
Data Structures, Function Implementations and 3–Address Translator	`ass6_`*roll*_translator.cxx
Target Translator and x86-64 Translator `main()`	`ass6_`*roll*_target_translator.cxx
Test Inputs	`ass6_`*roll*`_test<number>`.c
3-address Test Outputs	`ass6_`*roll*`_quads<number>`.out
Test Outputs	`ass6_`*roll* `<number>`.asm

5. Prepare a zip-archive with the name `ass6_`*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 tinyC Compiler: **10**