C to ARM Assembly Compiler Report

Xiaohan Chen
xc3913@ic.ac.uk

Overall Design and Structure:

I used Flex to generate token scanner and Bison to generate the grammar parser. Scanner passes token types and corresponding token contents to parser. Parser processes tokens individually and match them with specific grammar rules, otherwise return the syntax error message.

My parser includes one c++ header file which is used for building up the abstract syntax tree (AST). I wrote different classes (tree nodes) to describe different statement. They are all inherited from the "Value" class, which is beneficial to print the whole AST via different subclasses. These statement nodes will eventually stored into a class called FUNC_UNIT which is like a transfer station, store different types of statements into its members. Then the instances of this class are pushed into vector with type " FUNC_UNIT* ". This vector plays an critical role in keeping the sequence of lines of code in the program.

To be more specific, the instance of "DECLARATION" class will be created when the parser parses a line of code that is declaration, and it is put into "FUNC_UNIT" instance. Then when the parser returns back to the root, the instance is pushed back into the vector. The similar procedure is applied to other statements so the AST is form in this way. However, before the instance is pushed back, the semantics of the code is checked by the function "debug()" in the root class. After the push_back, semantic environment is updated by adding or deleting variables in the symbol table, which traces the existing variables.

The root instance is finally passed to "compiler.cpp" file as extern variable "g_ast". This file calls parser, then call the "Print()" function in the root instance to print out the AST or generate the ARM code.

Grammar:

my grammar starts with "program", it parses the int main function. Inside this production rule, there is "func_impl" which left-recursively parses the statements in the main program line by line including "declaration", "assignment" and "function call" etc. Each of the statements has its own production rule, but most of them make use of the production rule "expr" that is standing for expression. My expression consists of some basic elements like integers, variables or string, it also has some recursive operations like "expr OP expr", which could extend the operation into more than 2 operands, most importantly this kind of structure will form a binary tree of arithmetic operations. When it parses operations, I set the precedence of multiplication and division beyond that of addition and subtraction, so the expression will keep its original sequence of operations.

Design Decisions:

In parser, for expression, my previous grammar was :-

```
expr:
  factor
| expr op factor
;
```

In non-terminal factor, it contains all the basic nodes (e.g. integers or identifiers). However this will lose the lose the precedence of the operations because Bison is a bottom-up parser. For example, the operation “1+7*8” will be understood as (1+7)*8 by the parser, because the parser cannot shift anymore if it does not reduce “1+7” according to the previous grammar. So I changed it to something like:-

```
expr:
  Id
| Integer
| expr op expr
;
```

In my assembly code generation, I assigned specific registries for different roles. For instance, registries r0,r1,r3 are used for working out the arithmetic operations in the expression. r3 is also used for variable declaration and assignment. Therefore, there are massive amount of STR and LDM stack commands to store and load the data into and from the static stack, which is not efficient.

Algorithms or Data-structures:

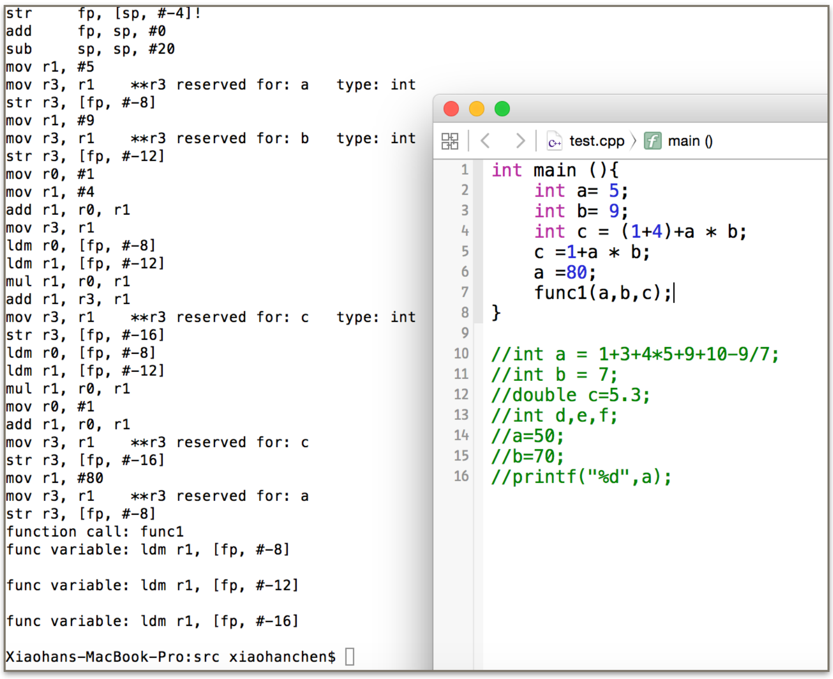
I created a map consisting of variables declared and their types for semantic check. This map is also necessary for the type checks and type conversion and so on.

I also made a vector which contains variables’ value, its index is corresponding to the address of static stack in assembly code, which is essential when the value of a variable is used either by declaration or function call etc.

In each statement class, I used different type numbers to distinguish different kinds of things. For example, in the “DECLARATION” class, I used type number 1 for the form of “int id = expr”, type 2 for “int a,b,c” and so on.

When I am generating assembly code, sequence of operations should be strictly followed, so I used recursive function call and control flow to output the correct arithmetic operations. Additionally, some special situations should be particularly coded. For instance, “(1+3) +(4*5)”, r0 is used for temporally storing the result of (1+3), so I cannot use r0 and r1 to load the value 4 and 5 anymore, in this case I will move the data in r0 to the r3, and keep using r0 and r1 to calculate the result of 4*5 as usually. This decision significantly reduces the amount of code modification and it could be kept for further development. And if there are variables in the expression, I need to use LDM instructions to pop out the value of that variable in the stack.

Output:



```
str      fp, [sp, #-4]!
add      fp, sp, #0
sub      sp, sp, #20
mov r1, #5
mov r3, r1     **r3 reserved for: a   type: int
str r3, [fp, #-8]
mov r1, #9
mov r3, r1     **r3 reserved for: b   type: int
str r3, [fp, #-12]
mov r0, #1
mov r1, #4
add r1, r0, r1
mov r3, r1
ldm r0, [fp, #-8]
ldm r1, [fp, #-12]
mul r1, r0, r1
add r1, r3, r1
mov r3, r1     **r3 reserved for: c   type: int
str r3, [fp, #-16]
ldm r0, [fp, #-8]
ldm r1, [fp, #-12]
mul r1, r0, r1
mov r0, #1
add r1, r0, r1
mov r3, r1     **r3 reserved for: c
str r3, [fp, #-16]
mov r1, #80
mov r3, r1     **r3 reserved for: a
str r3, [fp, #-8]
function call: func1
func variable: ldm r1, [fp, #-8]

func variable: ldm r1, [fp, #-12]

func variable: ldm r1, [fp, #-16]

Xiaohans-MacBook-Pro:src xiaohanchen$
```

```cpp
int main (){
    int a= 5;
    int b= 9;
    int c = (1+4)+a * b;
    c =1+a * b;
    a =80;
    func1(a,b,c);
}

//int a = 1+3+4*5+9+10-9/7;
//int b = 7;
//double c=5.3;
//int d,e,f;
//a=50;
//b=70;
//printf("%d",a);
```

My code is fully working on all types of the declaration and assignments. it is also able to detect most syntax errors occurring in these two statement.

Improvement:

I could work on the registry allocation by using the stamps to find out least recently used registry, then doing the STR or LDM on that registry to fully use all of other registries.

My variable table and static stack vector are for the declaration and assignment statement, consequently it would also be adequate for “if/while” statements or function calls if I am going to extend my compiler.

My grammar is also prepared for the extension of function declaration and global variable declaration before the int main function.

```
(
i.e.
program:
   func_decla_list SEMICOLM
   global_id_list SEMICOLM
   INTMAIN LBRACKET RBRACKET LCURLYBRACKET func_impl RCURLYBRACKET
   func_impl_list
{……… }
;
)
```