CSE 504: class project

E– Compiler

In this project, you will complete the rest of the E-- compiler. Your target language will be an abstract assembly language that is further described in Section 2. It is a 32-bit machine that has the following characteristics:

- 4 giga words of memory, where each word can store any value of primitive type such as float or int
- a large number of integer and floating point registers,
- supports move operations from memory to registers and vice-versa,
- supports conditional and unconditional branches (both direct and indirect branches), and arithmetic/logical operations (but not call/return),
- supports an operation to fetch the next event
- requires all operands to be in registers (except for move operations).

I am providing you the assembler to convert your abstract assembly code into machine code. It works by translating assembly code into C, and then uses gcc to translate this C-code into an executable. (Although this may seem convoluted, it was easy to implement and works well.) I am also providing a library that is responsible for delivering events to the code.

Completing this compiler involves the following tasks:

Type-checking: Implement a type-checker for E–. A description of the type checker and sample tests are provided to you. While the tests are designed to be helpful to you, you need not try to match the exact outputs shown in the sample output file; but use the sample outputs to ensure that your type checker works correctly.

Memory allocation: Allocate storage for global and local variables, as well as temporaries needed.

Implementing functions: Implement procedure calls and returns, as well as parameter passing. You should also make sure that recursion works correctly.

Compilation of event-matching: You can see that E– uses patterns on events that look very much like regular expressions. You need to compile these patterns into code for performing these pattern matches. Make sure that you test patterns that contain nontrivial combinations of closure, sequencing, and alternation operations.

(Intermediate) code generation: Generate abstract machine code.

Optimization: I expect you to implement some optimizations, although the exact ones are up to you.

We will make two modifications to E-- before proceeding on this assignment: we will add a while-loop, and a generalized form of break statement. While loops could be nested, and can contain break statements of the form "break n" that cause control to be transferred to the statement immediately following the end of the nth enclosing while-loop.

You will reuse the implementation of E– AST assignment 3 for this project.

1 Grading and Timeline

This project will be done by **groups of 3 to 5** students. Every group needs to assign a leader who will submit the project implementation on the Blackboard.

1.1 Grading

Unlike the programming assignments you have been working on so far, where the tasks were specified in great detail and with specificity, a project is much more "free form." I expect to see significant variation between groups in terms of what they deliver, and how well (and fast) it works.

Groups are expected to show a significant deal of imagination and creativity in their projects. I will award up to 10 bonus points for groups that surpass my expectations in this regard.

1.2 Timeline

Here are the deadlines:

3/16: Project posted.

3/21: Project groups and the leaders to be finalized, and the names sent to the TA.

4/12: Interim report due. By this date, you should have (mostly) completed the basic implementation.

TBD: (mostly between 5/13-5/20) Final report and project demos due. (will be announced)

You have around 2 months to complete the project, and 1 month to submit interim report. This is enough time to complete the project with good planning and execution. So plan properly, distribute tasks among team members, and talk to us in case of any questions or difficulties.

2 Target Machine

The intermediate code is intended to model a very simple instruction set. Complete information about the instruction set can be obtained from the source code files for the assembler. In particular, note that each icode instruction is translated by the assembler into a macro with the same name. The definitions of these macros appear in `E--_RT.c`, and should provide you a clear understanding of the instruction set semantics.

Icode supports 1000 integer registers (R000 to R999) and 1000 floating point registers (F000 to F999). The size of integers and floating point numbers is the same as the size of int and float types on `scm.oslab`.

Icode memory starts at address zero, and goes on to a maximum address specified as a command-line option to the C-program generated by the assembler. (The default memory size is 1M words.) Memory is word-addressed, not byte-addressed. This memory is represented as an array in the C-program. Guard zones are set up on either side of the array to catch out-of-bounds accesses (which will trigger a memory exception). To simplify the organization of memory, we make the assumption that `sizeof(int) == sizeof(float)`. This enables integers and floating point numbers to be both stored in one memory word.

Execution of icode starts at the first instruction in the icode file, and execution is stopped when control flows past the last instruction in the file.

The instructions can be divided into the following categories. In all cases, destination operands are listed following the source operands. Any instruction can have a label.

- *integer arithmetic and bit operations:* includes ADD, SUB, DIV, MUL, MOD, NEG, AND, OR and XOR. The last 3 operations are bit-wise operations, applying the specified boolean operation on corresponding bits of two integer operands. These operations have two source operands (just one source in the case of NEG) that can either be a value or a register, and a destination operand that must be a register.
- *floating point arithmetic:* includes FADD, FSUB, FDIV, FMUL and FNEG. They have two source operands (one in the case of FNEG) that can be values or registers, and a destination operand that must be a register.
- *integer relational operations:* includes GT and GE that treat their two operands as signed integers; UGT and UGE that operate on unsigned integers; EQ and NE that operate on integers regardless of size; All operands can be values or registers.

  NOTE: Relational operators are not stand-alone instructions, but instead, appear as part of a conditional jump instruction.
- *floating point relational operations:* includes FGT, FGE, FEQ and FNE that operate on floating point operands (values or registers).
- *print instructions:* include PRTI, PRTS, and PRTF that each take a a single operand. In the case of PRTI, this operand represents the integer value to be printed, or the register that needs to be printed. PRTF is similar, except that it prints a floating point operand. In the case of PRTS, the operand is a string constant (enclosed within double quotes) or a register containing a string constant, i.e., the register was previously initialized by a `MOVS <string_constant> <reg>`.
- *jump instructions:*
  - *unconditional jump:* `JMP <loc>`, where `<loc>` is a label.

  - *conditional jump:* `JMPC <cond> <loc>`, where `<cond>` is a relational operator with parameters. Example: `JMPC GE R000 R001 <loc>`
  - *indirect jump:* `JMPI <reg>`, where `<reg>` is an integer register that has previously been initialized with a label value using a MOVL instruction.
  - *conditional indirect jump:* Example `JMPCI FGE F001 1.0 R010`.

- *data movement instructions:*
  - *move label to a register:* `MOVL <label> <intreg>`
  - *move string to register:* `MOVS <stringConstant> <intreg>`
  - *move integer to register:* `MOVI <valueOrReg> <intreg>`
  - *move float to register:* `MOVF <valueOrReg> <freg>`
  - *move int to float reg:* `MOVIF <intreg> <freg>`
  - *move float to int reg:* `MOVIF <freg> <intreg>`
  - *load int reg from mem:* `LDI <reg> <valuOrReg>`
  - *load float reg from mem:* `LDF <reg> <valuOrReg>`
  - *store int reg to mem:* `STI <reg> <valuOrReg>`
  - *store float reg to mem:* `STF <reg> <valuOrReg>`

- *input instruction:* (for reading input data) `IN <reg>` reads a single byte from input stream and stores it into the specified register. A negative return value indicates an error, with the semantics the same as that of getc. `INI <reg>`, `INF <reg>` read an integer (or floating point number) into the specified register. Aborts execution if any errors are encountered.

3 Event Matching

In a full implementation of the E-- language, event matching should be implemented using one of the direct DFA construction techniques: either the one based on the concept of derivatives discussed in class, or the technique described in your textbook. Of course, neither of these algorithms handle event parameters. You can refer to Section 5 of the paper available at `http://seclab.cs.sunysb.edu/seclab/pubs/usenix99.pdf` to understand how event arguments can be handled. (Note that there are some notational differences: the paper uses ";", "||" and "∗" in the places of "`:`", "`\/`" and "`**`". The paper uses the term "REE" to refer to these event patterns, and NEFA to the automata constructed for matching these patterns.)

As mentioned before, you are not required to implement event patterns in this assignment. However, you may decide that you want to implement it any way. (If you do this, then you can skip some other component of E-- implementation, specifically, the implementation of functions and parameter passing. Moreover, you will not need to implement event arguments.)

Events will be input using the IN instruction in your assembly. You can assume that, for the purposes of this project, all event names will have only a single character, so it will be easy to input event names using the IN instruction that returns just a single byte. (This limits us to a maximum of 53 events.) For this assignment, we will restrict ourselves to integer and floating point event arguments. Since you have the declaration of events, which specifies the number and types of event arguments, you should be able to use an appropriate number of IN operations to input event arguments and convert them into integers or floating point numbers. For instance, given the following declaration

```
event a(int x1, float x2)
```

the input

```
aK\0\0\0\0\0\0AaP\0\0\0\0\0@A
```

will denote a sequence of two events `a(75,11.0),a(80,12.0)`. (Note that `\0` denotes a null character.) Note that the input representation reflects how integers and floating point numbers are represented internally. For instance, an integer is represented using 4-bytes, with the first byte representing the least significant byte of the integer. A similar observation applies to floating point numbers, except that their internal representation is a bit more complex. The above event information can be read using the following icode:

```
IN R100 // Read event name
....    // Decide how many parameters to read, and their types
INI R101 // Read the integer parameter
INF F101 // Read the floating point parameter
```

Note that `IN` behaves differently from `INI` and `INF` on errors: `IN` will return a negative value to denote input errors or end-of-file. The other two instructions will simply abort the program with an error message.

4 Project Path

Most of your implementation effort in this assignment will be concentrated in followings functions that you add to subclasses of AstNode and SymTabEntry, namely, `typeCheck`, `memAlloc` and `codeGen`. The former function is responsible for type checking. Second function is responsible for allocating memory: it will traverse the program to identify global variables and associate them with memory locations where they will be stored. The allocator will also need to determine the offsets on the activation records for parameters and local variables. Finally, it will need to compute the number of temporary variables needed, and allocate them. (Temporary variables need to be registers or local variables. If you use registers, then make sure that you do this only if you can bound the number of registers needed — for instance, if you have an expression that contained a few thousand operators, even 1000 registers would not be enough.)

Assembly code will be generated by the `codeGen` function.

You will need to construct appropriate test cases to test your code. Some of the key features that you should test are:

- Type checking: make sure that typing errors and coercions are handled correctly. Also, implicit and explicit typings are handled.
- Event pattern matching: Make sure that you test patterns that contain nontrivial combinations of closure, sequencing, and alternation operations.
- Function calls and returns: You should ensure that the code works with different types and numbers of parameters. You should also make sure that recursion works correctly.
- If-then-else, loops and break statements.
- Large expressions that test your technique for allocating and managing temporaries.

5 Submissions and Demo

We will follow the same submission requirements that we have been following so far for the assignments. The group leader will submit the project implementation on the Blackboard. He/she will mention the names and the IDs of the project members during submission. Project members do not need to submit anything to the Blackboard.

But note that most of the points for the projects are based on project demos. These demos will be scheduled within a few days after the day of the last lecture.

6 Example program in assembly code

```
JMP begin
// Test function
prt3: ADD R900 1 R901
LDI R901 R050 // Return address now in R050
ADD R901 1 R901
LDF R901 F051 // param 1
ADD R901 1 R901
LDI R901 R052 // param 2
ADD R901 1 R901
LDF R901 F053 // param 3
ADD R901 1 R901
LDI R901 R054 // param 4
```

```
PRTS R054
PRTF F053
PRTS "*"
PRTI R052
PRTS "="
PRTF F051
PRTS "\n"
ADD R900 5 R900
JMPI R050
//
// Here is where the main program starts
//
// Basic integer operations
begin: MOVI 1036 R000
MOVI 1172 R001
MOVI 1169 R002
SUB  R001 R000 R003
SUB  R001 R002 R004
ADD  R004 R003 R005
MUL  R003 R004 R006
DIV  R006 R004 R007
MOD  R001 R000 R008
NEG  R008 R008
// Basic FP operations
MOVF 1.036 F000
MOVF 1.172 F001
MOVF 1.169 F002
FSUB F001 F000 F003
FSUB F001 F002 F004
FADD F004 F003 F005
FMUL F003 F004 F006
FDIV F006 F004 F007
FNEG F007 F007
// Basic logical operations
MOVI 1036 R000
MOVI 1048 R001
MOVI 1052 R002
OR  R001 R000 R003
AND R001 R002 R004
XOR R001 R002 R004
// A simple loop
IN R010
ADD R010 1 R010
SUB R010 1 R020
IN R021
MOVIF R021 F022
FDIV F022 100.0 F022
IN R021
MOVIF R021 F021
FADD F021 F022 F021
MOVF 0.0 F011
L1: SUB  R010 1 R010
JMPC GE 0 R010 L2
FADD F021 F011 F011
JMP L1
// A simple function call
L2: PRTF F011
PRTS "\n"
MOVI 10000 R900 // init SP
MOVS "Computed using loop that " R022
STI R022 R900
SUB R900 1 R900 // pushed param 4
```

```
STF F021 R900
SUB R900 1 R900 // pushed param 3
STI R020 R900
SUB R900 1 R900 // pushed param 2
STF F011 R900
SUB R900 1 R900 // pushed param 1
MOVL L3 R025
STI R025 R900
SUB R900 1 R900 // pushed return addr
JMP prt3
L3: PRTI R900
PRTS "\n"
// Input two events: first arg of event is int, second is float
IN  R201
PRTI R201
PRTS "("
INI R200
PRTI R200
PRTS ", "
INF F200
PRTF F200
PRTS ")\n"
IN  R201
PRTI R201
PRTS "("
INI R200
PRTI R200
PRTS ", "
INF F200
PRTF F200
PRTS ")\n"
PRTS "\n***DONE***\n"
```