THE UNIVERSITY OF SYDNEY

COMP3109 Assignment 2

Compiler Front-End (10 marks)

This second assignment is a **group assignment** of up to three people, and is due in Week 9 **on Friday, September 26, at 5pm**. Your assignment will not be assessed unless all of the following criteria are met:

1. Hand in a signed academic honesty form in the tutorial before submission.
2. Submit a tarball of your source code to eLearning. The documentation should be done in form of comments. Please provide plenty of them.
3. You must include the `compile.sh`, `run.sh` and `README` files as described at the end of this assignment spec.
4. Your `README` file must list 1) the names of all group members, 2) the names and version numbers of all languages, compilers and tools used.
5. Your assignment must build cleanly with the listed compilers and tools. No errors, no warnings.
6. Your tarball must *not* contain any binary build products, such as executables or object files.

In this assignment you will be implementing the front-end of a small compiler and an interpreter for the intermediate code generated by the compiler. For ease of implementation, we recommend that you use scanner and parser generator tools such as Happy, Bison, ANTLR etc. The compiler can be written in any of C/C++, Java, Racket, OCaml, Scala or Haskell. For other languages check with your lecturer to ensure they can get an implementation.

The compiler should parse a given program, perform semantic checks, and then produce intermediate code. The interpreter can then be used to execute the intermediate code, which will simplify testing of the compiler. The grammars of the input language and intermediate code are defined below.

Advice

- Start early. It can take a long time to learn unfamliar tools such as scanner and parser generators, especially if you are not familar with the host language (eg Haskell).
- If you decide to use a particular tool like ANTLR or Bison, then change your mind, you will need enough time to redo the work using a different tool.
- Use a distributed version control system such as Git or Darcs to communicate between the group members.

- The *third* assignment in this course will build on the current one. If you take short cuts with code quality it will cause you pain later in the course.
- Choose group members of about the same skill level, and don't try to complete the whole assignment yourself.
- This assignment is easy to split into pieces. If you have extra time then use it to extend the source language, add more semantic checks, or improve the code generator.
- Exchange input code examples, as well as compiled code with other groups. You can then check if your implementation works with these examples. This is a good way to check for bugs in your assignment code. As this is a university assignment you should not exchange the *implementation code* itself, but exchanging input and intermediate code examples is fine.
- The first version of JavaScript was written by Brendan Eich in ten days. You have four weeks.

Input Language

Program Structure

A program consists of one or more function definitions, where each function has a name, followed by a list of arguments and a function body. A function body consists of an optional list of variable declarations and a code block. A code block finally contains a sequence of statements, which could either be (1) an assignment, (2) an if-then statement, (3) an if-then-else statement, or (4) a function return.

Assignments consist of a variable on the left hand side, which may either be an argument of the function or a variable declared before the function body. The right hand side of assignments can be arbitrary expressions, which may consist of signed integer numbers, variables (again including arguments and declared variables), function calls, and binary expressions (+,−, ...). Function calls consist of a function name, followed by a potentially empty list of variables, whose values will bit passed as arguments.

If-then, as well as if-then-else statements, consist of a variable as condition and one or two code blocks respectively. The then-block is only executed when the value of the condition is non-zero, the (optional) else-block is executed only when the condition has a value of zero.

Return statements take a variable as argument, whose value is returned to the calling function.

The language does not define any constructs for iteration other than recursion.

Semantic Rules

All variables must be declared before being used, i.e., before they appear in any statement. Variables are either declared as arguments to the function or using the `VARS` keyword within the function's definition. Locally declared variables and arguments reside in the same namespace, i.e., a name used for an argument cannot be used to declare a local variable (and vice-verse). Variables defined using the `VARS` keyword are implicitly initialized to zero. Functions are visible in the entire program and reside in a separate namespace, i.e., variables may have the same name. Names are case sensitive.

All variables, the values of expressions, as well as the return values of functions represent signed integer values.

Input Language Grammar

The following grammar specifies the syntax of the input language. The language is case sensitive, i.e., keywords must be in upper case.

```
<program>    ::= <functions>

<functions>  ::= ε
<functions>  ::= <function> <functions>

<function>   ::= FUNCTION <ID> <arguments> <variables> <block>

<arguments>  ::= ( ε )
<arguments>  ::= ( <id_list> )

<variables>  ::= ε
<variables>  ::= VARS <id_list> ;

<id_list>    ::= <ID>
<id_list>    ::= <ID> , <id_list>

<block>      ::= BEGIN <statements> END

<statements> ::= ε
<statements> ::= <statement> ; <statements>

<statement>  ::= <ID> = <expression>
<statement>  ::= IF <ID> THEN <block>
<statement>  ::= IF <ID> THEN <block> ELSE <block>
<statement>  ::= RETURN <ID>

<expression> ::= <NUM>
<expression> ::= <ID>
<expression> ::= <ID> <arguments>
<expression> ::= ( <expression> <OP> <expression> )

<NUM>        ::= -?[0-9]+
<ID>         ::= [a-zA-Z][a-zA-Z0-9]*
<OP>         ::= (+|-|*|/|<|>|==)
```

Intermediate Code

Program Structure

A program is a list of functions, where a function is a list consisting of the function's name followed by a lists of basic blocks. Each basic block in turn is a list consisting of a block number and a list of instructions.

Instructions

The intermediate code supports 13 different instructions:

- Load constant (`lc`)
  Copy a constant into a register.

  Example:
  `(lc r5 5)` ... copy the value 5 into register 5
- Load instructions (`ld`)
  Read the value of a variable and copy its value into a register.

  Example:
  `(ld r5 a)` ... read the value of `a` and copy it into register 5
- Store instructions (`st`)
  Write the value of a register to a variable.

  Example:
  `(st b r5)` ... read the value of register 5 and assign it to variable `b`
- Arithmetic instructions (`add`, `sub`, `mul`, `div`)
  Perform the respective arithmetic operations on registers.

  Example:
  `(add r3 r4 r5)` ... read register 4 and 5 and write the sum into register 3
- Comparison instructions (`lt`, `gt`, `cmp`)
  Perform a comparison and write the value 0 (false) or 1 (true) into a register.

  Example:
  `(cmp r3 r4 r5)` ... compare register 4 and 5 and write the result into register 3
- Branch instructions (`br`)
  Perform a conditional branch depending on a register value. If the value of the register is non-zero the execution continues at the basic block whose number is specified by the second operand. The execution, otherwise, resumes at the basic block specified by the third operand.

  Example:
  `(br r3 1 2)` ... depending on register 3 branch to basic block 1 or 2
- Return instructions (`ret`)
  Return the value of a register from a function.

  Example:
  `(ret r3)` ... exit the current function and return the value of register 3

- Call instructions (`call`)
  Perform a function call. The second argument specifies the function to be invoked, followed by a list of registers whose values should be passed as function arguments. The return value of the called function is written into the register specified as the first operand.

  Example:
  `(call r1 factorial r3)` ... invoke function `factorial` and pass the value of register 3 as its first (and only) argument; write the return value into register 1

Intermediate Code Grammar

```
<program>      ::= ( <functions> )

<functions>    ::= ε
<functions>    ::= <function> <functions>

<function>     ::= ( <ID> <arguments> <blocks> )

<arguments>    ::= ( <id_list> )

<id_list>      ::= ε
<id_list>      ::= <ID> <id_list>

<blocks>       ::= ( <NUM> instructions )
<blocks>       ::= ( <NUM> instructions ) blocks

<instructions> ::= <instruction>
<instructions> ::= <instruction> <instructions>

<instruction>  ::= ( lc <REG> <NUM> )
<instruction>  ::= ( ld <REG> <ID> )
<instruction>  ::= ( st <ID> <REG> )
<instruction>  ::= ( add <REG> <REG> <REG> )
<instruction>  ::= ( sub <REG> <REG> <REG> )
<instruction>  ::= ( mul <REG> <REG> <REG> )
<instruction>  ::= ( div <REG> <REG> <REG> )
<instruction>  ::= ( lt <REG> <REG> <REG> )
<instruction>  ::= ( gt <REG> <REG> <REG> )
<instruction>  ::= ( eq <REG> <REG> <REG> )
<instruction>  ::= ( br <REG> <NUM> <NUM> )
<instruction>  ::= ( ret <REG> )
<instruction>  ::= ( call <REG> <ID> <reg_list> )

<reg_list>     ::= ε
<reg_list>     ::= <REG> <reg_list>

<NUM>          ::= -?[0-9]+
<ID>           ::= [a-zA-Z][a-zA-Z0-9]*
<REG>          ::= r[1-9][0-9]*
```

Example

The following program implements the factorial function using the language specified above:

```
1  FUNCTION factorial(n)
2  VARS cond, tmp;
3  BEGIN
4    cond = (n == 0);
5    IF cond THEN
6    BEGIN
7      tmp = 1;
8      RETURN tmp;
9    END
10   ELSE
11   BEGIN
12     tmp = (n - 1);
13     tmp = (factorial(tmp) * n);
14     RETURN tmp;
15   END;
16 END
17
18 FUNCTION main(n)
19 VARS tmp;
20 BEGIN
21   tmp = factorial(n);
22   RETURN tmp;
23 END;
```

An example factorial program generated by the compiler is shown below:

```
( (factorial (n)
    (0   (ld r1 n)
         (lc r2 0)
         (cmp r3 r1 r2)
         (st cond r3)
         (ld r4 cond)
         (br r4 1 2) )
    (1   (lc r5 1)
         (st tmp r5)
         (ld r6 tmp)
         (ret r6) )
    (2   (ld r7 n)
         (lc r8 1)
         (sub r9 r7 r8)
         (st tmp r9)
         (ld r10 tmp)
         (call r11 factorial r10)
         (ld r12 n)
         (mul r13 r11 r12)
         (st tmp r13)
```

```
        (ld r14 tmp)
        (ret r14) ) )
(main (n)
  (0  (ld r1 n)
      (call r2 factorial r1)
      (st tmp r2)
      (ld r3 tmp)
      (ret r3) ) ) )
```

Task 1 Syntax and Semantics Analysis (2 marks)

Write a compiler front-end that parses a program as specified by the language above and performs a semantic analysis. The front-end should report the following error messages:

- Undefined function:
  Error: function '<function name>' undefined.
- Two functions with the same name:
  Error: function '<function name>' redefined.
- Mismatching number of arguments at a function call:
  Error: function '<function name>' expects <n> argument(s).
- Undefined variable:
  Error: variable '<variable name>' undefined.
- Two variables and/or function arguments with the same name:
  Error: variable '<variable name>' redefined.
- If a program does not define a `main`:
  Error: No main function defined.
- Syntax error:
  Syntax Error.

Task 2 Intermediate Code Generation (4 marks)

Extend the front-end from the previous task to generate intermediate code for those programs that do not contain any syntax or semantic errors. The intermediate code should be represented as S-expressions (LISP/Racket syntax) as specified by the intermediate language grammar above.

Basic blocks should be uniquely identified by a non-negative number, which can be freely assigned by the compiler. The compiler should generate at least one basic block for every block of the input language (`block` in the grammar of the input language) and break complex expressions down into individual instructions using registers to hold intermediate values. Sub-expressions of complex expressions can be evaluated in any order as long as data dependencies are respected. An unlimited number of registers is available to the compiler. You can assume that register values in the environment of a calling function are preserved across function calls, i.e., a new clean environment is created on a function call and the environment of the calling function remains unmodified until the function call returns.

Task 3 — Interpreter (4 marks)

Write an interpreter that can execute the intermediate code generated by the compiler front-end.

The interpreter operates on an environment consisting of a mapping of variables to values as well as a mapping of registers to values. The interpreter processes one instruction at a time, taking an environment as input and producing a new environment with the respective mapping of variables and/or registers updated according to the instruction description from before. The interpreter should be able to interpret across basic block boundaries as well as function calls. For a function call a new environment is created that is initialized with a mapping of the argument names to the respective value of the registers specified as arguments to the call instruction.

Reading a variable or register that is not associated with a value in the current environment should result in an error and the termination of the interpreter. Storing a value into a variable or assigning a value to a register should simply add a new or replace any existing mapping.

Function calls should supply a matching number of arguments to the called function. If this is not the case an error should be raised and the interpreter should terminate. The execution will then continue at basic block 0 within the new function.

The execution should start with a pre-initialized environment at basic block 0 of a function with name `main`. Returning from `main` causes the interpreter to print the return value before terminating. If no such function exists an error should be raised.

Example:
Given the initial environment `( ((a 71)) ((r3 0) (r4 5)) )`, the execution of the following instructions sequence results in the environments shown on the right (the environment after the instruction has completed is shown).

```
(ld  r3 a)                    ( ((a 71)) ((r3 71) (r4 5)) )
(st  a r4)                    ( ((a 5)) ((r3 71) (r4 5)) )
(add r5 r3 r4)                ( ((a 5)) ((r3 71) (r4 5) (r5 76)) )
(call r3 identity r5)         --
-- call identity              --
(ld r1 x)                     ( ((x 76)) () )
(ret r1)                      ( ((x 76)) (r1 76) )
-- return from identity       ( ((a 5)) ((r3 76) (r4 5) (r5 76)) )
```

Note that a new environment `( ((x 76)) () )` is created when function `identity` is entered.

Implementation

Your assignment should run on the `ucpu[01]` machines, though contact the lecturer *before submission* if this is not possible. You need to provide all code required to execute your compiler and interpreter. You need to provide at a minimum two scripts, `compile.sh` and `run.sh`, as well as a `README` file. The `README` should explain where the relevant code corresponding to each of the tasks can be found. The `compile.sh` script should take two arguments: the first argument is the name of the input file to compile, while the second argument is the name of the intermediate code file that should be generated. The `run.sh` invokes the interpreter, where the first argument specifies a file containing intermediate code should. Any additional arguments should be passed to the `main` function. Error checking should be done to ensure that the right number of arguments are supplied.