COMP304 Programming Languages (2016)
Assignment 3 : Programming Language Implementation
(Due midnight, Friday 26 August)

This assignment is intended to give you further experience in writing Haskell, and to increase your understanding of the design and implementation of programming languages by looking at how an implementation for a very simple programming language can be extended.

You are provided with an implementation for the very simple language of straight line programs with integer variables (`Straight.lhs`) discussed in lectures. The implementation consists of a compiler, which translates straight line programs into code for a simple stack-oriented virtual machine, and an interpreter for this stack code. For this language, the stack code only needs load and store commands, and commands for arithmetic operations.

You are asked to made several modifications which will transform this into an implementation for a more interesting language with nested control structures, variable declarations and procedures. This will involve making changes to the language being translated, the checks that are performed on it, and the stack code to which it is translated. In each part, you should ensure that the extended programs can be displayed in a readable format.

You will need to make a number of decisions about how to implement various features. You are encouraged to think about different options, and discuss them with the tutors, lecturer and other students (use the forum!) — and you should explain these choices as part of your submission.

Part 1: Adding control structures (30 marks)

First, you should extend the language to add **if** and **while** statements. We have seen in lectures how while programs can be represented using an algebraic data type. For this extension, you will need to add jump and conditional jump commands to the stack code and interpreter. At this stage, conditions should just be comparisons between integer values — this will be changed in Part 2.

The challenging part of this extension is in working out the destinations for jump and conditional jump commands. One way to do this, which is very similar to what is done in a real compiler, is to first give symbolic labels (which can be integers) to locations in the code, possibly using a new `Noop` command to attach labels to, and then translate these into absolute addresses (indexes into the list of commands) once the whole program has been translated.

Part 2: Adding Booleans, declarations and type checking (30 marks)

Second, you should extend the language to allow Boolean variables and Boolean expressions. This will mean that variables can hold Boolean values, and that expressions can return Boolean values, computed using Boolean variables, Boolean constants (`true` and `false`), and Boolean operators (`and`, `or` and `not`).

You should also add variable declarations, giving types (either `int` or `bool`) for variables, and impose the following constraints on programs:

(i) No variable may be declared twice.

(ii) Every variable used in the program must be declared.

(iii) Operators must be applied to arguments of the correct type.

(iv) Variables must only be assigned values of their declared type.

In the abstract syntax, a declaration can be treated as a list of type-variable pairs. A program will contain a single declaration, at the start.

Integer and Boolean expressions should now be represented using a single algebraic data type, and the types of variables, constants and operators used to determine the type of each expression.

This part should mainly affect the compiler. The only change to the interpreter will be to choose a way to represent Boolean values, and to implement Boolean operators. The checks made in the compiler should ensure that arithmetic, Boolean and comparison operators are always applied to values of the correct type.

Part 3: Simple procedures (20 marks)

Next, you should extend the language to allow simple procedures, with no parameters of local variables.

A procedure should have a name and a body. A program will now consist of a variable declaration part, and list of procedure declarations and a statement part. A statement in the body of the program, or of a procedure, may now be a procedure call giving the name of the called procedure.

There are two issues to think about in implementing simple procedures:

- What additional checks should the compiler make to ensure that the program is meaningful. Think about the checks made in Part 2, and how similar condition might be added for procedures.
- How to implement procedure calling and return. The code for a procedure will just be part of the code for the translated program, so to call a procedure you need to jump to the start of its code — so you need to have the address of the start of the code for a procedure available somewhere. On completion of the procedure, you need to return to the "next" statement after the procedure call, so the return address needs to be saved somewhere. In order to allow recursive calls, this should be saved on the run-time stack.

Part 4: Procedures with parameters and local variables (20 marks)

Finally, you should extend the language to allow procedures to have parameters and local variables.

A procedure declaration will now give a list of parameters (and their types) as well as the name and body, and the body will include a list of variable declarations, as already allowed for the main program.

Again, you will need to think about what additional checks should be performed to ensure that the program is meaningful. This will include thinking about the *scope* of variables (and parameters), since local variables of a procedure should only be able to be accessed within that procedure.

You will also need to think about what to do with parameters and local variables. The usual approach for parameters is to leave their values on top of the run-time stack when the procedure is called. You can also put local variables on the stack, or perhaps create some other kind of local store to keep them in — again, remembering that in recursive procedures, a new copy is required for each recursive call.

As an optional extra, you might allow procedures to return values and be called like functions within expressions, and allow variable declarations to occur anywhere in the body of of a program or procedure. But you should only try this after everything else is working correctly!

Submitting your work

You should submit your work via the submission system. Your submission should be in the form of a literate Haskell script, and **must** include an indication of which modifications you attempted and which of them you completed, explanation of how you implemented them, including what error conditions you check for and how, and discussion of any remaining problems.

Your script should include test cases illustrating the behaviour of your program, including incorrect behaviour if your program does not work correctly, and explanation of what each test case illustrates.

The marks shown for each part are indicative and may be varied. Within each part, around half of the marks will be awarded for writing good and correct code, about a quarter for explaining your design, and about a quarter for testing (giving test cases and explaining what happened).

Please read this handout again before you submit your assignment, and check that you have provided everything described above.

Total marks possible = 100.