COMP3012 - Compilers
Coursework

Venanzio Capretta / Nicolai Kraus

Release date: Wednesday 10 November 2021
Submission deadline: 5 January 2022
Updates made after 10 Nov are written in red

Contents

1 Description of the task

The goal of this coursework is to write a compiler for a simple imperative language called MiniTriangle (source: David Watt and Deryck Brown, *Programming Language Processors in Java*, 2000). In addition to the language of arithmetic expressions we've already studied, it has variables (containing integer values) and imperative commands for variable assignment, conditional instructions, while loops, and procedures to read and print integers on the terminal.

An example of a program written in the language is the following:

```
let var n;
    var x;
    var i
in
begin
  getint (n);
  if n < 0 then x := 0 else x := 1;
```

```
  i := 2;
  while i <= n do
    begin
      x := x * i;
      i := i + 1
    end;
  printint (x)
end
```

This program reads a number `n` from the terminal, computes its factorial and prints it in the terminal (if the input is negative, it prints `0`).

In the assignment instructions and in the Boolean test in conditionals and loops, we allow expression of the kind we already implemented, but we're allowed to use variables as well (`n < 0`, `x * i`, etc.).

2 Instructions, help, and how to get started

It is up to you how you wish to solve the coursework. Below are instructions and suggestions, but you can choose a different way than the one suggested if you want.

1. On the Moodle site for this module, under *Coursework*, you can find an archive `arithExp.zip` . This archive uses functional parsers to code a solution for the exercises of the previous lab sessions, i.e. it contains a compiler for the language of arithmetic expressions with boolean operators. There is a `README` file that documents the contents of the zip archive.

2. You are strongly encouraged to use functional parsers (and the theory we learned in the last couple of lectures) in order to solve this coursework. It is possible to completely avoid functional parsers and write scanner/parser in the "manual" way seen at the beginning of the lecture but this will result in repetitive code and significantly more work for you. Such solutions will be accepted for the coursework but may result in reduced marks for code quality. Before you choose to do this, please have a careful look at the solution in `arithExp.zip` to see the difference between the approaches.

3. The current coursework asks for a compiler for a richer language than the one in `arithExp.zip` . One approach would be to start from the compiler defined in that archive and extend it. If you wish, you can also start from scratch, but the library file `FunParser.hs` should be useful in any case.

4. Below, you find a precise specification of the language. Start with the parser. We will discuss the extension of the TAM language in later lectures.

5. Instructions on how to submit are given in Section 6 below. The submission deadline is 26 November. This is *Part A* of the coursework; a *Part B* is planned for later.

3 Grammar of the Language

The following is the grammar for (a fragment of) MiniTriangle, given in Backus-Naur Form. Non-terminals are in **bold**, terminals in `typewriter` font.

program	::=	`let` **declarations** `in` **command**
declaration	::=	`var` **identifier** | `var` **identifier** `:=` **expr**
declarations	::=	**declaration** | **declaration** `;` **declarations**
command	::=	**identifier** `:=` **expr**
	|	`if` **expr** `then` **command** `else` **command**
	|	`while` **expr** `do` **command**
	|	`getint` `(` **identifier** `)`
	|	`printint` `(` **expr** `)`
	|	`begin` **commands** `end`
commands	::=	**command** | **command** `;` **commands**

The non-terminal **identifier** (the syntactic category of name variables) denotes an alphanumerical string starting with a letter. This means that allowed variable names are strings such as `x`, `e83a`, or `ys`. However, keywords are reserved and are not valid identifiers. Thus, the strings `let`, `in`, `var`, `if`, `then`, `else`, `while`, `do`, `getint`, `printint`, `begin`, `end` are not valid variable names.

The grammar for expressions **expr** is the one we have implemented in the previous lab sessions; see also the implementation and comments in the archive `arithExp.zip`, which contains the grammar. However, we allow to use a `var` as an expressions (i.e. a **term** can now also be a `var` **identifier** and you need to modify the definition accordingly).

You will also need to extend the definition of Abstract Syntax Trees accordingly: every non-terminal should be associated to a type of ASTs for its syntactic category.

Note that a specification of indentations is not part of the language, i.e. the indentations used in the sample program in Section 1 are only there for readability.

4 Extension of TAM

We add to TAM new instructions that will allow us to translate the new commands.

We must be able to read and write to any location in the stack. We indicate positions in the stack by addresses of the form [n] where n is the location position with respect to the base of the stack. So the first cell in the stack has address [0], the second has address [1] and so on.

The TAM program is not executed sequentially any more, but we must be able to make jumps to implement conditional commands and loops. For this we must mark the places in the TAM code that we may jump to by labels. A label l can be any string. (Since the labels will be automatically generated by the compiler, we must have a mechanism to generate fresh labels: the easy way to do it is to use numbers.)

- HALT
  Stops execution and halts the machine
- GETINT
  Reads an integer from the terminal and pushes it to the top of the stack
- PUTINT
  Pops the top of the stack and prints it to the terminal
- Label l
  Marks a place in the code with label l, doesn't execute any operation on the stack
- JUMP l
  Execution control jumps unconditionally to location identified by label l
- JUMPIFZ l
  Pops the top of the stack, if it is 0, execution control jumps to location identified by l, if it is not 0 continues with next instruction
- LOAD a
  Reads the content of the stack location with address a and pushes the value to the top of the stack
- STORE a
  Pops the top of the stack and writes the value to the stack location with address a

5 Variable Environment

The values of the source program variables will be stored at certain addresses in the stack. Since the variables have to be declared at the beginning of the program (we have only global variables for the moment) we can assign to them the first few locations after the base of the stack.

While compiling the rest of the program, we must remember the address that we assigned to each variable. For this we need the code generator to have an extra argument: an *environment* consisting of a list of pairs (x, a) where x is a variable name (identifier) and a is a stack address.

6 Submission

Your compiler should be structured as a collection of Haskell modules with a `Main.hs` module.

Please look at the `README` in the archive `arithExp.zip` . It explains how, for the compiler in that archive, the main `Main` module can be used. Your submission should be usable in an analogous way.

In particular, it should be possible to compile your `Main.hs` with the command `ghc Main.hs -o mtc`.

The executable `mtc`, when called on a Triangle source file (`./mtc program.mt`) must generate TAM target code in `program.tam`. When called with a TAM program file (`./mtc program.tam`) it must execute it using the TAM virtual machine.

Submit your compiler as a compressed archive file (for example with the extension `.zip` or `.tgz`) with file name consisting of `compilers_cw1_` and your name and ID number, e.g. `compilers_cw1_venanzio_capretta_123456.tgz` or `compilers_cw1_nicolai_kraus_987654.tgz`.

Submission will be via the Moodle page. The deadline is 5 January 2022.