COMP3012 - Compilers
Coursework 1

Venanzio Capretta

Tuesday 29 October 2024

Write a compiler for the imperative language MiniTriangle.

In addition to the language of *arithmetic expressions* that you developed in the exercises of the previous lab sessions, it has: *variables* (containing integer values) and imperative commands for *variable assignment*, *conditional instructions*, *while loops*, and instructions to *read and print integers* on the terminal.

Submission

The compiler should be structured as a collection of Haskell modules with a `Main.hs` module to be compiled with the command:

```
> ghc Main.hs -o mtc
```

The executable `mtc`, when called on a MiniTriangle source file (extension .mt):

```
> ./mtc program.mt
```

must generate TAM target code written in file `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 (extensions `.gz` or `.zip`) with file name `compilers_cw1_`*your name and ID number*. For example my submission would be:

```
compilers_cw1_venanzio_capretta_123456.gz
```

Use the submission link on the Moodle page. The *deadline* for submission is *14 November at 3 PM.*

Important Restrictions: You can import only the following modules from the standard library:

```
Data.Char
Data.List
Control.Applicative
Control.Monad
System.Environment
```

No other module import is allowed.

No pragmas (compiler instructions) are allowed. These are lines in Haskell code looking like this:
`{-# .... #-}` **NOT ALLOWED**

Grammar of the Language

The following is a grammar for the MiniTriangle language. 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.

You will 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.

Expressions: The grammar for expressions **expr** is that of the Arith language, extended as described in the Lab exercise sheet:

We allow the use a `var` as an expressions (modify the definition of **term** accordingly).

We add relational operators (`<`, `<=`, `>`, `>=`, `==`, `!=`); Boolean operators (`&&` for conjunction, `||` for disjunction, `!` for negation); a conditional operator with three arguments: `b ? x : y` *(if b is true then return x, else return y).*

The operator precedence, from highest priority to lowest, is this: (1) `-` (unary negation); (2) `*`, `/`; (3) `+`, `-` (binary negation); (4) `<`, `<=`, `>`, `>=`, `==`, `!=`; (5) `!`; (6) `&&`; (7) `||`; (8) `_ ? _ : _`.

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.)

- LSS, GRT, EQL
  Binary operators corresponding to `<`, `>`, and `==`. (We don't add TAM instructions corresponding to `<=`, `>=`, `!=`; they should be realized in term of the other instructions.)
- AND, OR, NOT
  Operators corresponding to the Boolean operations.
- 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*

Example

An example of a program written in MiniTriangle 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, the result is `0`).

If this program is contain in a file `factorial.mt`, then compiling it with the command:

```
> ./mtc factorial.mt
```

will generate a file `factorial.tam` containing the compiled TAM program (the TAM program generated by your compiler may be slightly different, but it must execute correctly):

```
LOADL 0
LOADL 0
LOADL 0
GETINT
STORE 0
LOAD 0
LOADL 0
LSS
```

```
JUMPIFZ #0
LOADL 0
STORE 1
JUMP #1
Label #0
LOADL 1
STORE 1
Label #1
LOADL 2
STORE 2
Label #2
LOAD 2
LOAD 0
LSS
LOAD 2
LOAD 0
EQL
OR
JUMPIFZ #3
LOAD 1
LOAD 2
MUL
STORE 1
LOAD 2
LOADL 1
ADD
STORE 2
JUMP #2
Label #3
LOAD 1
PUTINT
HALT
```

When executing this TAM program with:

```
> ./mtc factorial.tam
```

We should have the following execution:

```
Executing TAM code
Enter a number:
6
output > 720
Final stack: [7,720,6]
```