COMP3012 - Compilers
Lab Sessions 2-4 - Exercises

Venanzio Capretta

Wednesday 9 October 2024

You will build a small compiler that you will extend progressively in the following weeks. You will have to submit as coursework an intermediate step and the final compiler for the full TRIANGLE programming language. You must start working on it immediately: if you fall behind, you won't be able to complete it in time.

Exercise 1 asks you to complete the compiler for the simple language of arithmetic expressions.

The following exercises ask you to extend the language with new features and modify the compiler accordingly.

1. Complete the interpreter and compiler for the language `Arith` of arithmetic expressions described in the lectures. You should write several Haskell modules to realize the various components: Scanner, Parser, Evaluator, TAM Virtual Machine, Code Generator; and a Main module for the the full compiler.

   When compiling Main with ghc, for example with the command:

   ```
   ghc Main -o aec
   ```

   you should get an executable `aec` (for *Arith Expression Compiler*) that both compiles files containing arithmetic expressions and executes TAM code, according to the extension of the file.

   If you run `aec` with a file with extension `.exp` (containing an arithmetic expression), it should generate a file with extension `.tam` containing the corresponding TAM code. An example source file `arith_example.exp` is given on the Moodle page. The command

   ```
   ./aec arith_example.exp
   ```

   should generate a file `arith_example.tam` containing the TAM code for the expression.

   If you run `aec` with a file with extension `.tam`, it should execute the TAM code and print the final result. The command

```
./aec arith_example.tam
```

should run the code generated by the previous command and print the result (33).

2. Extend the function `execTAM` (Lecture 4 notes) to a function `traceTAM` that prints the trace of the abstract machine computation:

   ```
   traceTAM :: Stack -> [TAMInst] -> IO Stack
   ```

   It should print a table showing the result of each instruction on the stack, finally returning the stack content after execution. For example, the command

   ```
   > traceTAM [] [LOADL 8, LOADL 3, DIV, LOADL 2, ADD]
   ```

   should print the table

   ```
   Initial stack:     []
   LOADL 8            [8]
   LOADL 3            [3,8]
   DIV                [2]
   LOADL 2            [2,2]
   ADD                [4]
   ```

3. Extend both the language of expressions and TAM instructions to include Boolean operations of conjunction, disjunction, and negation:

   In Expressions: binary operators `&&` (conjunction) and `||` (disjunction), unary operator `!` (negation). In TAM: `AND`, `OR`, `NOT`.

   We use integers to represent Boolean values, with False represented by 0 and True represented by 1 (by convention every non-zero value is considered True).

   Extend the compiler to include these operations.

4. Extend both the language of expressions and TAM instructions to include relational operations mapping two integers to a Boolean:

   In Expressions: `<`, `<=`, `>`, `>=`, `==`, `!=`. In TAM: `LSS` (corresponding to `<`), `GTR` (corresponding to `>`), `EQL` (corresponding to `==`).

   Extend the compiler to include these operations (We don't add TAM instructions corresponding to `<=`, `>=`, `!=`; they should be realized in term of the other instructions.)

5. Extend the language of expressions with a conditional operator with three arguments: `b ? x : y` *(if b is true then return x, else return y)* where b is a Boolean expression and x and y are either Booleans or integers. As

said before, all arguments are actually integers, but we assume that b is either 0 (False) or 1 (True).

The expression should evaluate to x if b is 1, y if it is 0 (we leave the semantics unspecified if b is different from 0 or 1; in line with the convention stated in Point 3, you may consider those values true and evaluate accordingly).

We don't extend the language of TAM: the new conditional operator should be realized in terms of the existing TAM instructions.

Hint: there are two ways of doing this exercise:

- After constructing the AST, containing an extra constructor for the conditional operator, transform it into an AST that doesn't contain the conditional operator, instead realizing it with arithmetic and Boolean operators (this is a simple example of an optimizer).
- Directly map the AST (including the conditional constructor) to TAM code, realizing the conditional with an appropriate sequence of TAM instructions.

[Both solutions are inefficient. A more efficient way can be done once we introduce conditional instructions in TAM.]

6. Modify the main program so it accepts options: If we compile the main program with

   ```
   ghc Main.hs -o aec
   ```

   then we can invoke it with either a `.exp` or a `.tam` file: A `.exp` file will be compiled to the corresponding TAM file; A `.tam` file will be executed and the result printed.

   The program should furthermore accept these options:

   - `--trace`: When executing a `.tam` file, trace it using `traceTAM` from Exercise 2.
   - `--run`: When called with a `.exp` file, it immediately executes the generated TAM code, instead of saving it to a .tam file
   - `--evaluate`: When called with a .exp file, it evaluates the AST using the function `evaluate`.