CST Part IB Supervisions

Example Sheet 3++

Petar Veličković

Lent Term 2016

Compiler Construction

If you managed to remain on top of the material, you should have already managed to attempt some of the material in *Example sheet 3*. You should submit it for this week, as before. From what I gathered the course has since then only focused on details of implementing a virtual machine. The VM is not a true "component" of a compiler in the strictest sense, and therefore the additional written work for this week will not be heavy, however there are a few interesting aspects we may wish to focus on.

1. The initial VM solution described in the course notes implies relying on the runtime environment of another language for simulating elements of the runtime environment of the language being interpreted. Explain the tradeoffs involved when creating such an interpreter.

2. Explain clearly the notions of a *stack*, *heap*, *stack pointer*, *frame pointer*, *stack frame* and *closure*, emphasising the ways in which the terms are related.

3. Define clearly the semantics of interpreting `MK_CLOSURE`, `APPLY` and `RETURN` instructions in the Jargon VM. Use diagrams to visualise the stack state before and after interpreting them.

4. Consider a very simple (but still Turing complete!) computer that has only a single instruction—`SUBLEQ A B C`, and a memory with 10000 locations. This instruction has semantics as follows:

   `SUBLEQ A B C` $\longrightarrow$ `if ((Memory[B] -= Memory[A]) <= 0) goto C;`

   We are tasked with writing a code-generating program (*effectively, a compiler*) for this machine that will be capable of evaluating arithmetic expressions consisting of constants, variables and $+, -, *, /$. **You do not need to write this program**.

   The variables will be given in locations `Memory[0] -- Memory[8]` and will be denoted by $M0 - M8$ in the input expression. Output should be stored in $M0$.

Assume that the expression is given in postfix notation, so there is no need to process operator precedence; e.g. input `M1 M2 + 8 *` corresponds to the expression $(M1 + M2) * 8$.

The program needs to output a list of `SUBLEQ` instructions calculating the desired expression and storing the result in $M0$; these instruction are loaded starting from memory location 9, and the program counter is initially set to 9.

When the program counter is at `x`, the command

```
SUBLEQ Memory[x] Memory[x+1] Memory[x+2]
```

will be called. If the `goto` gets invoked, then the program counter will become equal to `Memory[x+2]` at the end; otherwise, it will become `x+3`. If any of these locations are out of bounds, the program halts.

With all that in mind, answer the following:

a) What kind of *runtime environment* would you maintain in your generated `SUBLEQ` code, to simplify your computation as well as reduce the chances of corruption of the code by your maintained state?

b) What kind of *higher-level instructions* (representing an "intermediate form" of the final generated code) would you devise in order to make code generation easier (by breaking it up into smaller steps)? How would these higher-level instructions translate to `SUBLEQ` instructions only?

Hint: Consider first, for example, how addition could be implemented using the `SUBLEQ` instruction only.

5. (**OPTIONAL**) If you are feeling particularly keen[*], you may wish to implement the ideas discussed in the previous problem and test them on http://www.spoj.com/problems/ONEINSTR/ (most well-known languages are fully supported).

Practical work

We continue from the lexer onto constructing a recurisive descent parser for our language.

1. Add an additional rule: $Prog \longrightarrow S\ \boxed{\texttt{eof}}$, and consider $Prog$ to be the start symbol.

2. With this in mind, define a data structure/type in your language of choice which you consider to be most suitable for storing the output of the parser (namely, the parse tree of the provided program).

3. Rewrite the syntax of the language in order to avoid left recursion.

[*] **Word of warning:** this is *very difficult* to implement correctly, and unless several clever tricks are used when generating the code, you will be required to implement non-naïve algorithms for division and multiplication, which pose several complications on their own...

4. Implement a function `parse`, which takes as its input a list of tokens (as generated by `lex`) and produces an instance of the data structure you have defined in step 2.
5. Demonstrate that your function works as expected by presenting the result of lexing and parsing the program given in *Example Sheet 2*.
6. Propose approaches to making any generated error messages more useful to users.