Programming Language Concepts

CSCI-344 **Programming 8**
Term 20235

Interpreter Choice

1 Introduction

In this programming assignment, you will complete and/or extend one of the textbook interpreters in order to gain experience with the implementation of a language or language feature.

This programming assignment must be completed as a pair programming assignment; see [WK00] for useful guidelines on pair programming.

2 Description

Choose *one* of the following problems:

2.1 Type Inference for nano-ML

Complete Exercises 18 and 19 of Chapter 7 from *Programming Languages: Build, Prove, and Compare* (pp. 448 and 449). The exercises ask you to complete the implementation of type inference for the nano-ML interpreter written in Standard ML.

Source code for this problem is named `nml`.

Notes:

- The source code includes an example (`tsort.nml`) and its output (`tsort.soln.out`). It is a functional topological sort (written by Prof. Norman Ramsey), that makes a reasonably interesting test case. The `tsort.soln.out` output was generated by the following command with the reference solution:

  ```
  $ cat tsort.nml | ./nml -q
  ```

- Like with Programming 05: Type Systems, you can also check that your type inference implementation assigns correct types to the variables defined in the basis.
- Like with Programming 05: Type Systems, the real test of a type checker is not only that it accepts correct programs but that it rejects incorrect programs.

2.2 Unification and Non-logical Features for μProlog

Complete Exercises 38, 44, and 45 of Appendix D from *Programming Languages: Build, Prove, and Compare (Supplement)* (pp. S111 and S112). The exercises ask you to complete the implementation of constraint solving for and to add implementations of the `!` (cut) and `not` predicates to the μProlog interpreter written in Standard ML.

Source code for this problem is named `uprolog`.

Notes:

- When implementing `!`, you may find it helpful to review Section 2.10.2 from *Programming Languages: Build, Prove, and Compare*, which discusses the use of continuation-passing style for backtracking search.
- The reference solution and tests from Programming 07: Prolog Programming should serve as a good test of the implementations of substitution and unification. However, they do not use `not` or `!` (cut).

2.3 Mark-Sweep Garbage Collector for μScheme

Complete Exercises 7, 8, and 9 of Chapter 4 from *Programming Languages: Build, Prove, and Compare* (pp. 294 – 296). The exercise asks you to complete the implementation of a mark-sweep garbage collector for the μScheme interpreter written in C.

Source code for this problem is named `uscheme-ms`.

Notes:

- You should only need to modify `ms.c`.
- You will find it helpful to read Appendix N from *Programming Languages: Build, Prove, and Compare (Supplement)*, which describes the supporting code for garbage collection in the μScheme interpreter written in C.
- The source code includes an example (`eval.scm` and `evaltest.scm`) and sample output (`eval_evaltest.soln.out`). It is the metacircular evaluator described in Section E.1 as well as a couple of short tests of the metacircular evaluator. It should trigger a fair number of garbage collections. The `eval_evaltest.soln.out` output was generated by the following command with the reference solution:

  ```
  $ cat eval.scm evaltest.scm | ./uscheme-ms -qq
  ```

- The reference solution and tests from Programming 03: Scheme Programming should serve as a good test.

2.4 Copying Garbage Collector for μScheme

Complete Exercises 1 and 2 of Chapter 4 from *Programming Languages: Build, Prove, and Compare* (p. 293). The exercises ask you to complete the implementation of a copying garbage collector for the μScheme interpreter written in C.

Source code for this problem is named `uscheme-copy`.

Notes:

- You should only need to modify `copy.c`.
- You will find it helpful to read Appendix N from *Programming Languages: Build, Prove, and Compare (Supplement)*, which describes the supporting code for garbage collection in the μScheme interpreter written in C.
- The source code includes an example (`eval.scm` and `evaltest.scm`) and sample output (`eval_evaltest.soln.out`). It is the metacircular evaluator described in Section E.1 as well as a couple of short tests of the metacircular evaluator. It should trigger a fair number of garbage collections. The `eval_evaltest.soln.out` output was generated by the following command with the reference solution:

  ```
  $ cat eval.scm evaltest.scm | ./uscheme-copy -qq
  ```

- The reference solution and tests from Programming 03: Scheme Programming should serve as a good test.

2.5 Method Caches and Class Variables for μSmalltalk

Complete Exercise 41 of Chapter 10 from *Programming Languages: Build, Prove, and Compare* (pp. 724 – 725). The exercise asks you to add method caches to the μSmalltalk interpreter written in Standard ML.

In addition, add support for class variables to the μSmalltalk interpreter. Class variables are discussed briefly in Section 10.12.2 (paragraph *More variables: Class instance variables and class variables*; p. 704). The concrete syntax for class definitions becomes:

def ::= `(class` *subclass-name*
`[subclass-of` *superclass-name*`]`
[`[cvars` {*instance-variable-name*}`]`]
[`[ivars` {*instance-variable-name*}`]`]
{*method-definition*}`)`

As described in the text, class variables are accessible to all the instances of a class and its subclasses, as well as the class itself and its subclasses. (To avoid confusion, if x is declared as a class or instance variable of an ancestor, it may not also be declared as a class or instance variable of class C.)

Source code for this problem is named `usmalltalk`.

Notes:

- For the first part (method cache), the reference solution and tests from Programming 06: Smalltalk Programming may make a good test.
- For the second part (class variables), the source code includes an example (`cvars.smt`) and sample output (`cvars.soln.out`). The `cvars.soln.out` output was generated by the following command with the reference solution:

  ```
  $ cat cvars | ./usmalltalk -qq
  ```

- When changing the syntax of the language, you will find it helpful to read Appendix I from *Programming Languages: Build, Prove, and Compare (Supplement)*, which describes scanning and parsing (i.e., the conversion from concrete syntax to abstract syntax).

2.6 Quasiquotation, `list`/`set-car!`/`set-cdr!`, and Rest Arguments for μScheme

Complete Exercises 55, 56, 58, and 59 of Chapter 2 from *Programming Languages: Build, Prove, and Compare* (pp. 197 and 198). The exercises ask you to add support for quasiquotation to the μScheme interpreter and to add `list`, `apply`, `set-car!`, and `set-cdr!` to the initial basis; although Chapter 2 describes the μScheme interpreter written in C, for this problem, modify the μScheme interpreter written in Standard ML.

In addition, add support for rest arguments to the μScheme interpreter and upgrade appropriate functions (either primitive or pre-defined; e.g., `+`, `<`, `and`, `max`) from the initial basis to accept an arbitrary number of arguments (even zero). Rest arguments are briefly described in the antepenultimate (third to the last) paragraph of Section 2.14.1 (p. 169); they are the mechanism by which Scheme functions can take a variable number of arguments.

Source code for this problem is named `uscheme`.

Notes:

- When changing the syntax of the language, you will find it helpful to read Appendix I from *Programming Languages: Build, Prove, and Compare (Supplement)*, which describes scanning and parsing (i.e., the conversion from concrete syntax to abstract syntax).
- Adding `list` as a primitive is a simple exercise in modifying the interpreter, but after implementing rest arguments, it has a trivial solution as a pre-defined function.
- Since this is a change to the existing language, there aren't readily available tests. However, the changes are backwards compatible, so you can test that your updated interpreter continues to work with existing uScheme programs (such as the reference solution and tests from Programming 03: Scheme Programming).

3 Interpreter Source Code

Source code for the various problems is available on the CS Department file system at:

`/usr/local/pub/mtf/plc/programming/prog08-interp-choice/`*`problem-name`*

and packaged as an archive at:

`/usr/local/pub/mtf/plc/programming/prog08-interp-choice/`*`problem-name`*`.tar`

Note that the source code contains just the C code or Standard ML from the textbook, with simple comments identifying page numbers. There is a `Makefile` for building the interpreter.

Copy the interpreter source code to a local directory and make modifications to your local copy; for example, executing

```
$ tar xvf /usr/local/pub/mtf/plc/programming/prog08-interp-choice/nml.tar
```

will copy the interpreter source code to a new local directory named `nml`.

4 Requirements and Submission

Modify the chosen interpreter as necessary.

Your modified interpreter must be a valid C or Standard ML program. In particular, it must compile with gcc or with Moscow ML or MLton without any error messages. If your submission produces error messages (e.g., syntax errors or type errors), then your submission will not be tested and will result in zero credit for the assignment.

Write a `README.txt` file. Your `README.txt` file should be formatted as follows:

```
Name(s):
Project Choice:
Time spent on assignment:
Additional Collaborators:

... description of your solution to the problem ...

... description of how you tested your submission ...
... (why are you convinced that it works correctly) ...

... description of your submission's functionality ...
... (what is working, what is not working) ...
```

In essence, the `README.txt` file should include a narrative description of the work done to complete this programming assignment. It will carry substantial weight when awarding partial credit.

Submit all modified files and `README.txt` to the `Programming 08` Dropbox on MyCourses by the due date. Only one submission is required per group.

References

[WK00] Laurie A. Williams and Robert R. Kessler. All I Really Need to Know About Pair Programming I Learned in Kindergarten. *Communications of the ACM*, 43(5):108–114, May 2000.