COMP 50 Lab: Serialization Via Stack Machine (The Compiler)

December 4–5, 2013

The theme of this lab is *how things work under the hood.* When you leave the lab, you will have a little bit of intuition for how data are stored on disk and for how writing an S-expression to disk works.

Data definitions: stack programs

An *atom* is one of

- A string
- A number
- A symbol

An *sx*, also known as S-expression, is one of

- An atom
- A `(listof sx)`

A `program`, also known as *stack-machine program*, is one of

- `empty`
- `(cons 'ATOM (cons x previous))`
  where `x` is an atom and `previous` is a program
- `(cons 'EMPTY previous)`
  where `previous` is a program. Note that the data contains the *symbol* `'EMPTY`, which is not the same as the empty list.
- `(cons 'CONS previous)`
  where `previous` is a program. Again, the data contains the *symbol* `'CONS`, as well as an application of the *function* cons.

A *stack* is a `(listof sx).`

What's happening

The idea of a `program` is that it is *a sequence of instructions for building data structures*. To enable us to use natural recursion, programs are written right to left. Unless you read Hebrew or Arabic or manga, you'll find right-to-left order strange, but bear with it—after Thanksgiving we'll learn a new design recipe that will make it possible for us to read programs from left to right.

We write an *interpreter* for the program, which manipulates a *state*. For us the state is a stack of values, which is another name for a list:

```
;; interpret : program -> state
;; to produce a stack of values according to the instructions in the program
```

Interpreting the empty program produces an empty stack. When a program is not empty, here's how each instruction is interpreted:

- To interpret an `'ATOM` instruction, we interpret the previous instructions, take the resulting stack, and push the atom `x` onto it.
- To interpret an `'EMPTY` instruction, we interpret the previous instructions, take the resulting stack, and push the empty list onto it.
- To interpret the `'CONS` instruction, we interpret the previous instructions, take the resulting stack, take the first two elements off that stack, apply `cons` to them, and push the result onto the stack.

What does this have to do with the real world?

Here are the connections:

A. For the trigrams project, you convertes your list of models to an `sx`, and DrRacket wrote the `sx` out to disk. The S-expression is a relatively high-level data structure.

B. On the disk, all information is stored as a sequence of bytes. DrRacket does the conversion from S-expressions to bytes.

C. In this lab, you'll "compile" an S-expression to a sequence of *atoms*. (Every `program` is also a `(listof atom)`.) The sequence of atoms is at a high enough level that you can do it in one lab, but at a low enough level that you will have some idea how it might work with a sequence of bytes.

D. You're used to writing programs in BSL and ISL. These programs have rich structure which makes them good for thinking and problem-solving. But hardware doesn't know a thing about BSL or ISL, and DrRacket has to translate from BSL or ISL to something the machine understands. This translation is a ton of work.

E. What the machine actually understands is a *sequence* of *machine instructions*, which are themselves represented as bytes. If you go on to COMP 40, you'll learn about machine instructions and how they work.

F. In this lab, we have very simple programs: a program is a sequence of instructions, and there are only three instructions. I've written the interpreter; you'll write a compiler. They will be powerful enough to do something useful but simple enough that you can make progress in the lab.

The lab problems

Solve the following problems:

1. Here is a program that first puts an empty list on the stack, then puts 2, then makes a cons cell, then puts 1 on the stack, then makes a cons cell, then finishes.

   ```
   '(CONS ATOM 1 CONS ATOM 2 EMPTY))
   ```

   The program is interpreted from right to left! When the program is interpreted, it should produce a stack containing exactly one value:

   ```
   (list (cons 1 (cons 2 empty)))
   ```

   This stack is the result of computing the *value* `(cons 1 (cons 2 empty))` and pushing it onto an empty stack.

2. Write a *compiler* that is given an S-expression and produces a program that, when interpreted, reconstructs the original S-expression. (This is how my function `write-file-sexp` works.)
3. *Extend* the data definition for `program` so that you can take three values from the stack and make a 2D-point.
4. *Extend* your definition of `sx` to include `(2Dpoint sx)`.
5. *Extend* my interpreter and compiler to support 2D-points.
6. Write a function that tests to be sure that every compiled S-expression is a list of atoms.
7. *Extend* our *system* (data definitions, interpreter, compiler, tests) so you can save and restore a full `(2Dtree sx)`.

Your results

Once you master the techniques of this lab, you'll be able to save many forms of data to disk. Here's the general technique:

> To save a data structure, write a *program* which, when interpreted, reconstructs the original data structure.

Once you master this technique you can use it again and again for whatever projects you work on.

Submitting the lab

Five minutes before the end of the lab, put the following text at the beginning of a DrRacket file. You may use an empty file or the source code you have been working on:

```
#|
What I did during this lab:
   (you fill in this part)

What I learned during this lab:
   (you fill in this part)

|#
```

Finally, *submit this file through the handin server* as `lab-final`. **Submit it as lab, not as homework.** You will need to submit it using *two* usernames connected by a + sign, as in

```
Jane.Doe+Richard.Roe
```

You submit using Jane Doe's password.