# Homework 4: Operational Semantics and λProlog

Due: November 9, 2004.

In this homework, you will apply your skills in logic programming to the study of operational semantics. Operational semantics are widely used in programming language theory. We will be using these techniques in our study of aspect-oriented programs which is coming up shortly.

This is an individual homework. That is, for this homework, you are to do the work on your own, not in groups.

For all λProlog programs, you must run your code with a λProlog system, such as the Teyjus system we recommend. You must also provide evidence that your program is correct (for example, test cases). Hand in a printout of your code and your testing.

You may use cut (`!`), `not`, and `once` in your code. (But don't get in to the mode of trying to sprinkle cut throughout your code to make it work; cut should be used sparingly and you should think about what you're doing instead.)

The problems in this homework all deal with an example in λProlog that we will discuss in class. The code is found in the `hw4` subdirectory of our course homework directory, accessible from the course web page, and also from the directory

```
/home/course/cs541/public/homework/hw4/
```

on the department machines.

## 1 While Language Expressions

1. (20 points) This problem is to add a new expression that increments a location; this is similar to the (prefix) `++` operator of C and C++. To do this, in the module `while_syntax`, add the syntax for a new expression (not a new command or operator) of the form `inc` $L$, where $L$ is a location. The intended meaning is that `inc` $L$ increments the value stored in location $L$ by 1, and returns this final value. In the module `while`, add new semantic rules to specify this operation. (Hint: use `access0` to access the store.) Finally, add tests to the module `while_test` to test the new code, run them.

   Hand a printout of the `while_syntax` signature, the `while` and `while_test` modules, and output from testing.

2. (10 points) The `inc` expression has side effects. Explain how the code in the `while` module is set up so that these side effects are not lost, but are reflected in subsequent evaluations. Give examples (which could be tests for the previous problem) as part of your explanation.

## 2 While Language Commands

3. (20 points) Add a do-until loop to the languge, so that `do` $C$ $E$, executes $C$, and then tests $E$; if the value of $E$ is 1, then the command is finished, but otherwise it repeats, executing `do` $C$ $E$ again.

   Hint: can a do-loop be desgared into other constructs in the language?

   Hand a printout of the `while_syntax` signature, the `while` and `while_test` modules, and output from testing.

4. (40 points; extra credit) Consider adding `break` commands, as in C, C++, and Java, that break out of the smallest enclosing while or do-until loop. The syntax is easy, but the trick for this problem is showing how to give their semantics in the "little-step" style. Hint: if it's

useful, you can introduce abstract syntax for commands that do not correspond to anything in the user-visible concrete syntax to help with this. You might add a command like `loop` $C$, as an aid to jumping out of the loop body when the `break` command is encountered. But you have to think about what happens when `break` commands are embedded in abstract syntax trees like `(semi (semi break` $C_1$`)` $C_2$`)`.

5. (100 points; extra credit) Is is possible to handle labels and `goto` commands in such a little step operational semantics? If so, show how; if not, explain why it can't be done.

## 3 Richer Domains, adding Booleans

6. (60 points) Currently, the expressible and storable values are only integers. To add booleans as a separate kind of value, make a new module `value` which defines two types: `value` and `boolean`. The value module should be accumulated by the `while_syntax` module. There should be two constructors for values: `intval` and `boolval`. There should also be constructors for the type `boolean`, `tt` and `ff`, representing true and false values (respectively). (Note that λProlog uses `true` as a built-in predicate already, so we can't use that.)

   Also, add to the expressions (in `while_syntax`'s signature). First, add a boolean literal expression, `bool` $B$, where $B$ is a boolean value. Also, add the short curcuit expressions `and` $E_1$ $E_2$ (conjunction), and `or` $E_1$ $E_2$ (disjunction), and `neg` $E$ (logical negation), where $E$ and $E_i$ are expressions. Add expressions `lt` $E_1$ $E_2$, which only works for integer arguments and returns a boolean, and `equals` $E_1$ $E_2$, which compares either two integers or two booleans, and returns a boolean.

   Change the definition of the language semantics in the module `while` as necessary. First change the declarations in `while.sig` so that `store int` is replaced with `store value`; this difference allows locations to store both integer and boolean values. Also change the types of the predicates that work with expressions so that they return `value`s, not `int`s. In the module itself you'll have to make corresponding changes. In particular, change the `if` and `while` commands so that they require a test expression to return a boolean value. Then add the semantics for the new expressions. (Hint, if you get the message "cannot decide consistent type for overloaded operator" add an explicit typing to one of the arguments, as in `I1:int < I2`.)

   When adding expression semantics, first decide what the terminal expression configurations are (ones to which no reductions apply). Then work on each new piece of syntax separately. If the expression isn't part of a terminal configuration, then write down the reduction rule when the subexpressions are reduced as far as possible. Then write rules for evaluating nonterminal subexpressions from left to right. To do this, look at the three op rules for an example. You will need several rules for each new expression.

   Adjust the types in `while_test_helpers`, and revised the test cases and expected outputs in `while_test` for these changes. (Note that the lists representing stores will now be lists of values, and that you'll have to modify the tests for `while` and `if`.) Get the existing tests to work again, then add tests for the new expressions.

   Hand a printout of the new `value` module, and other changed modules. You should not need to change the `store` module, as that is already polymorphic in the things it stores. Also hand in output from testing.

## 4 Envrionments

7. (50 points) In this problem you'll add environments, which are mappings from identifiers to locations, to the language semantics. We will use strings as the domain of identifiers.

   Syntactically, it's easiest to make a change to the abstract syntax, so that `id` $S$, where $S$ is a string, is considered to have type "location". Not, however that we are still using integers as locations in the store. The syntax, `id` $S$, is just an abstract syntax tree in the syntactic domain of locations, not a semantical location. That is, in the signature of module `while_syntax`, change (or add) the following constructor for locations.

```
type id     string -> location.
```

To add envrionments to the semantics, you can use the module `env`, which is provided for you. We will be using environments of type `(environ string int)` to map from names (represented by strings) to semantic locations (integers).

To change the semantics, it seems best (for grading) to make a fork in the development files. Make a copy of the files for the module `while` to create a module `while_env`. This module will accumulate the `env` module. In the signature, change the types of `reducesE` and `reducesC` to

```
type reducesE (environ string int)
               -> configurationE -> configurationE -> o.
type reducesC (environ string int)
               -> configurationC -> configurationC -> o.
```

You should also change the types of `meaningE` and `meaningC` to pass an environment as the first argument. Then change the rules in the `while_env.mod` file to reflect the changed types and the new syntax. Note that environments are not part of configurations. When using `rtc`, you should apply it to `(reducesC Env)` and `(reducesE Env)`, where `Env` is some envrionment, as this will make the types work out. Change all the rules to pass the environment to each rule, and either change or add new rules to deal with the new syntax for location abstract syntax trees (of the form `id` $S$). In particular, the semantics of assignment commands and dereference expressions need to be changed.

Finally, change the test aparatus, by making a copy of `while_test_helpers` in a new module named `while_test_helpers_env`, changing the types as necessary, and making a copy of `while_test` in a new module named `while_test_env`. You'll find it helpful to introduce a predicate of the form

```
type test_env (environ string int) -> o.
```

that is a "hook" for a test environment to be used by `runC` and `runE`. The rule(s) for this predicate should go in the `while_test_env` module. Change all the tests to work with `id` instead of `loc`. Add any additional tests you feel helpful.