Programming Language Concepts

Documentation

● Matthew Consterdine
● Alex O’Callaghan

Introduction

A C-like imperative language, for manipulating infinite streams. Statements are somewhat optionally terminated with semicolons, and supports both block (`/* ... */`) and line comments (`// ...`). Curly brackets are used optionally to extend scope. Example code can be found in Appendix A, quirks can be found in Appendix B.

Before continuing, it's helpful to familiarise yourself with Extended BNF. Special sequences are used to escape.

Usage Instruction

Once the interpreter has been compiled using the `make` command, you can choose to run an interactive REPL or a stored program. Executing `./mysplinterpreter` with no arguments will start in an interactive REPL . You should save your program files as `<filename>.spl` and pass the location as the first argument to ./`mysqlinterpreter`. As data is read from standard in, you can pipe files in using the `<` operator, or pipe programs in using the `|` operator, allowing you to create programs that manipulate infinite streams.

Starting the interactive REPL	`./mysplinterpreter`
Executing a saved program	`./mysplinterpreter <file> [ < <input> ]`
Using infinite streams	`<program> | ./mysplinterpreter <file>`

Programs are executed in multiple stages:

1. Entire program is loaded into a string (Unix line endings required)
2. Program is lexed into Tokens
3. Tokens are parsed into an Abstract Syntax Tree (Left associative with the exception of `lambda`s)
4. Types are checked to ensure logical behaviour
5. Finally the Abstract Syntax Tree is executed

Types

Our programming language has 4 basic types: `unit`, `int`, `pair`, and `lambda`. From these types we can create the complex data types: `bool`s with a simple comparison to 0, `list`s from `pair`s, and `string`s from `list`s. In addition `lambda` definitions can be nested to allow both currying and multiple parameters [Appendix B].

Our language is strongly, statically typed. Types are defined using the following Extended BNF grammar:

```
type     = unit | int | pair < <type> , <type> > | <type> -> <type>
```

Values

Complex data types are automatically decomposed into the basic types before they can be used in the program.

```
unit     = ?[]?
int      = [ 0 ( b | o | x ) ] { <digit> | _ }            (* Bin/Oct/Dec/Hex *)
pair     = ?[? <value> . <value> ?]?
lambda   = lambda ?(? <type> <identifier> = <value> ?)? <expression>
bool     = true | false
list     = ?[? <value> { , <value> } ?]?
string   = " [ { <character> } ] "
```

Variables

Variables are used to store dynamic values, with assignment similar to most C-like programming languages. To allow scoping, variables are implemented and type checked using two separate environment trees. As a result, you can access and manipulate variables higher in the tree, and variables that leave scope are lost.

Identifiers must be at least 1 character, start with a letter, and lexed using: `[a-zA-Z][a-zA-Z0-9-_]*`

```
bind     = <type> <identifier> = <value>
lookup   = <identifier>
rebind   = <identifier> ( = | += | -= | *= | /= | %= | &= | ^= | ?|?= ) <value>
unbind   = unbind <identifier>
```

Math

Like most C-like languages, our language supports all common unary and binary `math` operators:

```
unary    = ( ! | + | - ) <value>
binary   = <value> ( + | - | * | / | % | ^ | & | ⊻ | ?|? ) <value>
```

Also, various `math` related functions are built in (You can also call the above with `math.plus` for example):

```
math. (random | min | max | abs | sign | sqrt | ln | log | fact | ... ) …
```

`random` takes a number of parameters. Zero = bool, One = between 0 and value, Two = between the values.

Input/Output

Our programming language provides many inbuilt functions for manipulating standard input/output/error.

```
input    = console.read_ ( int | string | bool )
output   = console.print [ ln ] _ ( int | string | bool ) <value>
error    = console.error [ ln ] _ ( int | string | bool ) <value>
clear    = console.clear
```

The `int`, `string`, or `bool` instructs the interpreter on how to interpret the value it receives. For example:

```
raw = 65             int = 65             string = "A"            bool = true
```

Conditionals

Conditionals evaluate an expression if another evaluates to `true`. In our programming language `0` and `unit` evaluate to `false`, and every other value evaluates to `true`. This allows nice concise comparisons.

```
if       = if <condition> then <consequent> [ else <alternative> ] done
ternary  = <condition> ? <consequent> : <alternative>
coalesce = <condition> ?? <default>
compare  = <value> ( > | >= | < | <= | == | != ) <value>
```

Any expression can be used as the condition for a conditional, not just a comparison.

Loops

Our programming language implements the 3 traditional `loop`s any programmer has come to expect, plus `loop` which is equivalent to `while true`. Use `break`, `continue` and `return` to manipulate flow.

```
loop     = loop <expression>
while    = while <condition> do <expression>
do       = do <expression> while <condition>
for      = for <bind> ; <condition> ; <action> then <expression>
```

If you wish to use multiple expressions inside the `loop`, you can add matching curly braces.

Functions

Once a function has been created using a `lambda` expression, you can apply a value to it, to execute the code stored within. The syntax for doing so is very simple: `<lambda> ?(? <value> ?)?`. This can be chained if there are nested `lambda`s. A side effect of this allows you to create a nested `lambda` and curry it.

```
int -> int times10 = lambda (int value) (value * 10); times10 (20);
```

Lists

Our programming language has inbuilt support for Scheme-like `pair`s and `list`s. You can use `cons`, `head`, and `tail` to build and access the `pair`s. These `pair`s can be used to easily create a tree structure: `[[1 . 2] . [3 . 4]].` Alternatively to build `list`s, you can use a nice shorthand just like in Scheme:

```
(cons 1 (cons 2 (cons 3 [])))  [1 . [2 . [3 . []]]]  [1, 2, 3, []]  [1, 2, 3]
```

In addition, our language provides a number of inbuilt functions allowing you to manipulate `list`s, functionally.

```
map      = list.map    <lambda> <list>
fold     = list.fold   <lambda> <list>  <accumulator>
filter   = list.filter <lambda> <list>
limit    = list.limit  <list>   <value>
length   = list.length <list>
```

`string`s are simply lists of `int`s, therefor you can use any of the above to manipulate them, or `string.lower <list>`, and `string.upper <list>` to manipulate case and `string.rev <list>` to reverse it.

Miscellaneous

You can use `assert <value>` and `exit` to halt (un-)conditionally in your program.

Error Messages

End of File   Our program only accepts files with Unix style LF file endings. Use your text editor to convert.
Syntax Error   Input cannot be lexed into tokens, the program will give a line and column number.
Parse Error   Tokens cannot be parsed into an Abstract Syntax Tree, the program will give the incorrect tokens.
Type Error   Error while typing, the program will give a descriptive message.
Eval Error   Error while executing, the program will give a descriptive message.

Appendix A - Example Code

Hello World

```
console.println_string "Hello, World!";
```

Factorial

```
int -> int factorial = lambda (int n) {
     int result = 1;
     for int i = 1; i <= n - 1; i += 1 then
          result *= i;
     result;
};

console.println_int factorial (5);
```

FizzBuzz

```
for int i = 0; i <= 99; i += 1 then {
     if i % 15 == 0 then
          console.println_string "FizzBuzz"
     else if i % 3 == 0 then
          console.println_string "Fizz"
     else if i % 5 == 0 then
          console.println_string "Buzz"
     else
          console.println_int i
     done done done
}
```

Guessing Game

```
console.println_string "I'm thinking of a number between 1 and 10 (inclusive)";
int number = math.random 1 10;
int guess  = console.read_int;

if number == guess then
  console.println_string "You guessed correctly!"
else
  console.println_string "Better luck next time!"
done;
```

Functional List Operations

```
console.println_int list.filter (lambda (int x) x) [1,2,3,0];
console.println_int list.fold (lambda (int x) (lambda(int y) (x+y))) [1,2,3] 1;
console.println_int list.limit [1,2,3] 2;
console.println_int list.map (lambda (int x) (x+1)) [1,2,3];
```

Appendix B - Language Quirks

Nested Functions

Below is an example of nested `lambda`s and currying, it allows you to create functions for `y = m * x + c`:

```
int -> int -> int -> int y = lambda (int x) (
     lambda (int c) (
           lambda (int m) (m * x + c)));
int -> int f = y (2) (5);
console.println_int f(10);
```

Unfortunately due to a typo (a single character) the type checker fails to apply the second value and throws an error stating that it expected a function but was given a number. This can be fixed with the patch below:

```
diff --git a/Checker.ml b/Checker.ml
index 14c69e6..e92bd6a 100755
--- a/Checker.ml
+++ b/Checker.ml
@@ -88 +88 @@ let rec typeOf env e = flush_all(); match e with
-                               | true -> tT
+                               | true -> tU
```

Fortunately the problems can be easily solved without the use of nested `lambda`s. This is not a resubmission.

Functional List Operations

Due to time constraints, you are limited in what you can do with the output of the functional list operations. While the output of `fold` can be used everywhere, the output of `filter`, `limit`, and `map` can only used in conjunction with `console.print [ ln ] _int <value>`. You cannot chain functional list operations.

For Loops

`for` is split up into 4 sections: `for <bind> ; <condition> ; <action> then <expression>`

In most languages such as C and Java, `<bind>` is executed, then for every iteration `<condition>` is checked, `<expression>` executed and finally `<action>` is executed. In our language `<action>` and `<expression>` swap places, which means the value in the initial `<bind>` and the `<condition>` may need to be tweaked to get the intended behavior. Alternativly the patch below will give the for loop standard behaviour.

```
diff --git a/Evaluator.ml b/Evaluator.ml
index 5e3b3b6..6329f91 100755
--- a/Evaluator.ml
+++ b/Evaluator.ml
@@ -180,2 +180,2 @@ let rec eval co ce ci env e = flush_all (); match e with
-                                       ignore (eval co ce ci scope c);
-                                       ignore (eval co ce ci scope d)
+                                       ignore (eval co ce ci scope d);
+                                       ignore (eval co ce ci scope c)
```