Assignment 3

Algorithms: Correctness and Complexity

Due 2017-Nov-14 10:30AM sharp.

The work that you turn in for this assignment must represent the effort of each group of one or two students. You are welcome to help your fellow students understand the material of the course and the meaning of the assignment questions, however, the answer that you submit must be created by one group alone.

John McCarthy (September 4, 1927 – October 24, 2011) is most famous for inventing the term Artificial Intelligence in the mid 50s and then being a leading researcher in the field for the rest of his life. He also made significant contributions to algorithmic languages (Algol), functional programming (LISP), time sharing and interactive operating systems (CTSS), program semantics, and verification. McCarthy was the first person to make use of Alonzo Church's λ-calculus in programming language design. This assignment is based on LISP.

For this problem set, you will complete a compiler for a higher-order, dynamically-typed, lexically-scoped, functional/imperative language called EOL (Essence of LISP). EOL is basically simplified LISP with 'nicer' (in my opinion) syntax.

The language:

Like LISP, EOL is an interactive language. The user types in a series of definitions and expressions. The system responds by printing the value of each expression and definition.[1]

$$\begin{aligned} \text{Start} &\longrightarrow \textbf{bye} \mid \textbf{eof} \mid \text{Item Start} \\ \text{Item} &\longrightarrow \text{Def ;} \mid \text{Expr ;} \mid \text{;} \\ \text{Def} &\longrightarrow \textbf{let } id := \text{Expr} \end{aligned}$$

Definitions introduce new top-level variables and give them initial values.

```
1? let a := 21 ;
21.0
2? a * 2 ;
42.0
```

The data values of the language fall into one of five data types:

- Numbers, e.g. 3.1415, 42, -1.

[1] Throughout, I'll use the abbreviation

$$n \longrightarrow \alpha \mid \beta$$

to mean that there are two productions for n, namely

$$n \longrightarrow \alpha \text{ and } n \longrightarrow \beta$$

and so on for 3 or more productions

Here **eof** refers to the actual end of the file, **bye** and **let** are keywords, *id* represents any identifier that is not a keyword.

- Strings, e.g. "hello world"
- The empty list: [ ]
- Cons pair values, which include nonempty lists: e.g. [1,"hello"]
- Function values (also called closures).

Binary operators include assignment (which changes the value of an existing variable), equality (=), cons (:), addition (+), subtraction (−), multiplication (∗), and division (/). Equality is nonassociative; cons is right associative; addition, subtraction, multiplication and division are left associative.

$$\begin{array}{rcl} \text{Expr} & \longrightarrow & id := \text{Expr} \mid \text{RTerm} \mid \text{RTerm} = \text{RTerm} \\ \text{RTerm} & \longrightarrow & \text{CTerm} \mid \text{CTerm} : \text{RTerm} \\ \text{CTerm} & \longrightarrow & \text{Term} \mid \text{CTerm} + \text{Term} \mid \text{CTerm} - \text{Term} \\ \text{Term} & \longrightarrow & \text{Factor} \mid \text{Term} * \text{Factor} \mid \text{Term} \,/\, \text{Factor} \end{array}$$

E.g.

```
3? a := 50 ;
100.0
```

The empty list is written `[]`. In EOL, the empty list doubles as the representation of falsity. Any other value represents truth; the built-in operations result in the string “true”.

```
5? 1=2 ;
[ ]
6? a=a ;
“true”
```

The colon (or cons) operator constructs a pair out of its two operands:

```
7? let p := "a" : 2 ;
"a" : 2.0
```

The head and tail operators take these cons pairs apart.

```
8? hd p ;
"a"
9? tl p ;
2.0
```

Cons pairs are used to represent lists as follows. `[]` is the empty list. Lists of length 1 are (represented by) cons pairs in which the tail is the empty list

```
10? 3 : [] ;
[3.0]
```

Lists of length 2 are (represented by) cons pairs in which the tail is a list of length 1, etc.

11? `2 : [3.0]`
[2.0, 3.0]
12? `2 : (3 : [] )`
[2.0, 3.0]
13? `1 : (2 : (3 : []) ) ;`
[1.0, 2.0, 3.0]

The : operator is right associative, so the preceding expression can be written as `1 : 2 : 3 : []`; it can also be written as `[1,2,3]`.

Operations on lists include `hd` (which gives the first value of a list), `tl` which gives the list of all values other than the head, and the predicate `null` which reports whether a value is the empty list or not.

14? `let b := [1,2,3] ;`
[1.0, 2.0, 3.0]
15? `hd b`
1.0
16? `tl b`
[2.0, 3.0]
17? `hd tl b ;`
2.0
18? `null []`
"true"

The predicate `atomic` is true of values other than cons pairs.

$$\begin{aligned} \text{Factor} &\longrightarrow \textbf{hd}\ \text{Factor} \\ \text{Factor} &\longrightarrow \textbf{tl}\ \text{Factor} \\ \text{Factor} &\longrightarrow \textbf{null}\ \text{Factor} \\ \text{Factor} &\longrightarrow \textbf{atomic}\ \text{Factor} \\ \text{Factor} &\longrightarrow \text{Call} \end{aligned}$$

$$\text{Call} \longrightarrow \text{Primary}$$

The simplest expression are identifiers (which, as we have seen, evaluate to the value of the variable), numbers, strings (written between double quotes)

$$\begin{aligned} \text{Primary} &\longrightarrow \textbf{id} \\ \text{Primary} &\longrightarrow \textbf{num} \\ \text{Primary} &\longrightarrow \textbf{string} \end{aligned}$$

We also have if expressions, which evaluate to either the first or second block, parenthesized blocks,

and list expressions.

Primary	$\longrightarrow$	**if** Expr **then** Block **else** Block **end if**
Primary	$\longrightarrow$	(Block)
Primary	$\longrightarrow$	[ ]
Primary	$\longrightarrow$	[Expr MoreList
MoreList	$\longrightarrow$	, Expr MoreList |]

Each block is a list of one or more definitions and expressions separated by semicolons. The value of the block is the value of the final expression or definition.

Block	$\longrightarrow$	Block1
Block1	$\longrightarrow$	Def MoreBlock | Expr MoreBlock
MoreBlock	$\longrightarrow$	; Block1 | ϵ

Blocks establish local scopes (or environments). I.e., the variables introduced within a block are local to that block.

```
19? a := (let c := 20 ; c+1) ;
21.0
20? c ;
Virtual Machine Error: Fetch from unknown variable: "c"
```

Functions: An interesting aspect of EOL is that functions are first-class values. (Function values are sometimes known as 'closures'). Syntactically we have expressions —traditionally known as a lambda expressions— which evaluate to function values

Primary	$\longrightarrow$	Function
Function	$\longrightarrow$	{ Params -> Block }
Params	$\longrightarrow$	OneOrMoreParams | ϵ
OneOrMoreParams	$\longrightarrow$	**id** MoreParams
MoreNames	$\longrightarrow$	, OneOrMoreParams | ϵ

Functions can be assigned to variables:

```
21? let square := {x -> x*x } ;
closure
```

Functions can be called using a fairly standard syntax:

Call	$\longrightarrow$	Call Args | Primary
Args	$\longrightarrow$	(ExprSeq)
ExprSeq	$\longrightarrow$	Expr MoreExprSeq | ϵ
MoreExprSeq	$\longrightarrow$	, Expr MoreExprSeq | ϵ

For example:

```
22? square( 6 ) ;
```

36.0

Parameter passing is "by value".

Functions can be used as arguments to other functions

```
23?
let map := {f, xs ->
   if null xs then [ ]
   else f(hd xs) :  map(f, tl xs)
   end if };
closure
24? map( square, b ) ;
[1.0, 4.0, 9.0]
25: map( {x -> x*x*x}, b ) ;
[1.0, 8.0, 27.0]
```

Study this example closely; note how the value of `square` is passed to the parameter `f`.

It is even possible to return a function as the value of a function

```
26? let incr := {x -> {y -> x+y } } ;
closure
27? incr(4)(5) ;
9.0
28? let i1 := incr(1) ;
closure
29? i1(5) ;
6.0
29? map( incr(10), b ) ;
[11.0, 12.0, 13.0 ]
```

The result of `incr(1)` is a function that is assigned to variable `i1`. Notice that: during an execution of function `id1`, `x` has the value 1.0; whereas, during an execution of the result of `incr(10)`, `x` has the value 10.0. This is because function values carry with them the environment (i.e. the mapping from identifiers to values) in which they were created. Each expression is evaluated in the context of a chain of environments. Variables are looked up in this chain starting from the top of the chain. When `i1(5)` is evaluated, the expression `x+y` is evaluated in a chain of environments as follows

- At the top of the chain is an empty environment. Function bodies are blocks and a block introduces a new environment, whether you need it or not.
- Next is an environment mapping the parameter y to 5.
- Next is another empty environment; this is from the outer function body.
- Then an environment mapping the parameter x to 1.
- Finally there is the global environment which maps, a, b, map, incr, incr1, etc.

The last 3 items of this chain were captured when `incr(1)` was evaluated.

This way of treating variables is called *lexical scoping*, as any variables mentioned in an expression need to be declared in some scope that lexically includes the expression.

Because of the assignment operator, environments are actually mutable mappings. This is what makes the language imperative as well a functional. The combination of lexical scoping and assignment allows for some object-oriented like effects.

```
30? let counter := { -> let x := 0 ; {x -> x := x+1 ; x } } ;
closure
31? let c := counter() ;
closure
32? let d := counter() ;
closure
33? c() ;
1.0
34? c() ;
2.0
35? d() ;
1.0
```

Each invocation of `counter` creates a distinct environment and so a distinct `x` variable. This environment is captured as part of the function value that `counter` returns. We can say that `counter` is something like a class and its results are something like objects.

The mutability of environments is also useful for support of recursive "anonymous" functions. Consider

```
let factorial := {n -> if n=0 then 1 else n*factorial(n-1) end if } ;
```

At the time the lambda expression is evaluated, the `factorial` variable does not exist. However by the time the body needs to be evaluated, `factorial` has been introduced and defined.[2]

About the virtual machine.

The machine code targeted by the compiler is defined by its implementation in class Machine. Details of the instruction set can be found in the internal documentation of the InstructionSet interface and the code and comments in class Machine.

The virtual machine is a stack machine. In fact it has three stacks: a value stack, a chain of environments, and a stack of unfinished function invocations. Additionally the state of the machine contains a program memory, which is an array of segments, where each segment is an

[2] Miraculously, we can create recursive routines without resorting to assignment or self-reference. See the Wikipedia article on fixed-point combinators.

```
? let fact := {g -> (g(g)) }(
   {f -> {n ->
       if n=0 then 1
       else n*f(f)(n-1) end if } } ) ;
? fact(5) ;
120.0
```

Note that the function `fact` is self contained. There is nothing circular about the definition.

This ability of a mathematical objecct to be self-referential without expliciitly referring to itself is the key to several important results in logic; most famously, Kurt Gödel used a similar trick to design a sentence whose meaning was essentially "this sentence is not provable". This showed that all logics sufficiently powerful to express such sentences are either incomplete —if the sentence is true, there is at least one sentence that is true, but not provable— or inconsistent —if the sentence is false, there is a sentence that is provable and false.

array of bytes. A program counter and a segment number together identify the next instruction to be executed.

Evaluation of each expression (or definition) leaves the result on the top of the value stack. In particular

- The CONST instruction pushes a new value on to the stack. The operand of this instruction is an index into a "constants table". Thus to compile a constant (string or double) one should first "inter"[3] the constant in the constants table and then emit the instruction CONST(i), where i is the index of the constant in the constants table. By convention, item 0 of the table is Nil and item 1 is the string "true".
- Unary operations replace the top of the stack with the result.
- Binary operations replace the top two items on the stack with the result.

Variables are stored in the machine's environment chain.

- As with the CONST instruction, the operand of the FETCH instruction is an index into the constants table. When executed, the FETCH(i) instruction finds a string constant in the constants table at index i, and then searches for that string in the environment chain. Thus to compile a variable look-up, the variable's name should be interred and a FETCH instruction emitted.
- STORE(t), similarly expects to find a string (variable name) in the constants table at index t. It searches for the string in the environment chain and modifies the value of the variable in the environment to be the same as the top of the stack.
- The NEW_VAR(t) creates (unless it already exists) a new variable in the topmost environment in the environment chain. Then, it behaves just like STORE(t). Neither NEW_VAR nor STORE alter the value stack.

Function values (closures) are made using the MAKE_CLOSURE instruction. Function values are invoked using the CALL instruction.

- Function values are created by a MAKE_CLOSURE(s) instruction. For example `{x,y->` $\cdots$ `}` could be translated into 3 CONST instructions —to push "x", "y", and 2 onto the stack— followed by a MAKE_CLOSURE(s) instruction. The operand s is the location of the machine code for the body of the function. The instruction memory is segmented and it is assumed that each function body starts at location 0 of some segment. Closures are represented internally by triples (π, η, s) where π is the list of parameters, η is the environment the closure was created in, and s is the segment number for the function's body.
- The following calling convention is used: First the closure to be called is placed on the stack, next come the arguments (rightmost last). Finally comes a CALL(n) instruction where n is the number of arguments. So a call `f(a, b)` is coded as three FETCH instructions followed by a CALL(2). CALL(n) works like this:
  - n values are popped off the stack to form an argument list α.

[3] Interring a constant returns its index in the constants table, modifying the table if the constant does not already occur there.

  - The closure (π, η, s) is popped off the stack.
  - It is checked that the number of arguments equals the number of parameters.
  - An environment $\eta' = \{\pi_0 \longmapsto \alpha_0, \pi_1 \longmapsto \alpha_1, \pi_2 \longmapsto \alpha_2, \cdots\} \rightarrow \eta$ is created (i.e., an environment that maps the parameters to the arguments and that is chained to the environment that was current when the closure was constructed).
  - The current program counter, segment, and environment are pushed on the stack of unfinished function invocations.
  - Execution resumes with instruction 0 of segment s and current environment set to η'.

- RETURN instructions simply restore the program counter, the current segment, and the environment from the stack of unfinished function invocations.

You can output instructions to standard output as they are being emitted if you turn on the machine's verbose flag. You can trace the execution of the machine code using the machine's tracing flag. These are set in the main program.

About the compiler: The compiler is written in the JavaCC 5.0s language. In JavaCC, Java code is interspersed with an extended-context-free grammar. The JavaCC code is in a file called `EOLParser.jj`. The JavaCC compiler generates from this file a number (7) of Java classes, including EOLParserTokenManager, which is a lexical analyzer, and EOLParser, which is a recursive-descent parser, and which also contains the main program. *Do not modify the generated files!*

What you need to do: The virtual machine is finished. The compiler is almost finished. Unfortunately the compiler guru had to be let go after exercising both her right to free expression and her middle finger at the same time. You need to complete the compiler. All that is left is to add some code to the parser to recognize and generate code for some expression forms. You need to modify the EOL.jj file (and only that file) so that all expressions of the language compile correctly. Pay attention to precedence and associativity of operators.

There is set of unit tests ready to go. If you create more unit tests, please submit them too.

You do not have to worry about syntax errors, as JavaCC will automatically generate code to throw an exception when a syntax error is encountered. Your compiler should not do type checking. Nor should it check that variables that are used have been declared. These checks are made at run time.

Submitting this assignment: Complete EOL.jj file. Put your name(s) in the comments at the top, and submit using D2L. Only the EOL.jj file needs to be changed; only the EOL file needs to be submitted.

Information on JavaCC. JavaCC extends CFGs in a number of ways, notably:

- It allows Java code to be interspersed in the grammar. This code is simply incorporated into the resulting method. Java code is included in braces.
- Nonterminals can have parameters, can have local variables, and they can return results.
- Only one rule is allowed per nonterminal, but the right hand side of the rule can be considerably more sophisticated than a simple sequence of terminals and nonterminals:
  - $\alpha \mid \beta$ means a choice between α or β.
  - $[\alpha]$ means a choice between α and nothing (ϵ).
  - $(\alpha)^*$ means a choice between ϵ, α, $\alpha\alpha$, $\alpha\alpha\alpha$, etc.
  - $(\alpha)^+$ means a choice between α, $\alpha\alpha$, $\alpha\alpha\alpha$, etc.

- JavaCC will generate code to decide these choices on the basis of the next token of input. When it can't, a "choice conflict" warning is given and you should either rewrite the grammar (usually preferable), or use a LOOKAHEAD specification to help JavaCC make the choice (usually not preferable).

You can download JavaCC 5.0 from

https://javacc.org/download

If you use Eclipse, you can instead install the Eclipse plugin for JavaCC from

http://eclipse-javacc.sourceforge.net/

There is a tutorial on JavaCC at

http://www.engr.mun.ca/~theo/JavaCC-Tutorial/

There is a FAQ at

http://www.engr.mun.ca/~theo/JavaCC-FAQ/

A non-JavaCC tutorial on parsing expressions by recursive descent is at

http://www.engr.mun.ca/~theo/Misc/exp_parsing.htm

The second method (the classic method) in the tutorial is the easiest to use.