MP 2 - Parsers and Interpreters

CS421 Agha - Spring 2015

rev. 2.91

1 Introduction

Assigned: February 11, 2015

Due: 11:59pm, February 27, 2015

Outline: This assignment will account for 8% of your grade.

2 Instructions

Please name your solutions script `mp2.scm` and submit as described in

`https://wiki.cites.illinois.edu/wiki/display/cs421sp15/Assignments?src=contextnavchildmode`

Please have your solutions in the same order as listed here. Some of these problems may require you to write additional procedures not mentioned explicitly. Make sure you include `#lang eopl` at the top of your script.

Note that the code discussed in class is available at: `https://github.com/mwand/eopl3`. In particular, look at the interpreter code for Chapter 3.

3 Academic Honesty

You may consult the web or talk about **concepts** with other students but if you use any code not written by you, *you need to cite the source in comments*. Failure to do so is considered plagiarism which is a serious infraction. You may also use any code from the book (check scheme resources on the wiki) but document that as well. Adapting code you found on the web will not be penalized as long as it constitutes less than 33% of your solution without penalty–as long as the comments for the specific code include a cite or an acknowledgement to your source (person, web page, repository) with the code.

4 Problems

In the grammars below, terminals are in courier font, for example, `lambda`. Nonterminals are italicized, as in *Expr*. Parentheses are literals (i.e. part of the specified grammar), while braces are part of the meta-language we use to describe the grammar. We use the * symbol to denote zero or more repetitions; for example, {(*Expr Var*)}* means zero or more occurrences of (*Expr Var*), including the parentheses. When it is clear from context, we omit the braces, so that *Var** means zero or more repetitions of *Var*. Also, recall that *Var*? means "zero or one occurrence" of *Var*.

We want you to cater for syntax errors (all exercises), and unbound variables (exercise 3). In these cases, you should output an error message. Use `(eopl:error "error message here")`.

1. Take a look at page 18 of the "data abstraction" lecture slides. Write an interpreter for this language. You may assume the source program is given as a Scheme list as opposed to a string of characters. For reference, the grammar you are parsing is:

Expression	::=	*Variable*
	::=	(`lambda` (*Variable*) *Expression*)
	::=	(*Expression Expression*)

   To make sure our test scripts work on your code, name your interpreter routine `lambda-interpret`. Example output:

```
> (lambda-interpret '((lambda (x) x) x))
'x

> (lambda-interpret '((lambda (x) x) y))
'y

> (lambda-interpret '((lambda (x) (lambda (y) x)) x))
#procedure or data structure representing a prodedure

> (lambda-interpret '(((lambda (x) (lambda (y) x)) x) y))
'x

> (lambda-interpret '((lambda (x) (lambda (y) x)) z))
#procedure or data structure representing a prodedure

> (lambda-interpret '(((lambda (x) (lambda (y) x)) z) y))
'z

> (lambda-interpret '(((lambda (x) (lambda (y) y)) x) y))
'y

> (lambda-interpret '((((lambda (test)
                           (lambda (then-clause)
                             (lambda (else-clause)
                               ((test then-clause) else-clause))))
                                  (lambda (x) (lambda (y) x))) a) b))

'a
```

```
> (lambda-interpret '((((lambda (test)
                            (lambda (then-clause)
                              (lambda (else-clause)
                                ((test then-clause) else-clause))))
                                   (lambda (x) (lambda (y) y))) a) b))


'b
```

A word of caution: Observe that unlike Scheme variables, free variables in the lambda-calculus will not denote a value: in the lambda-calculus, when a free variable x is used, it will evaluate to itself. In Scheme, a variable has a binding it evaluates to.

2. Write an interpreter for the following language of infix operators.

Concat-expr	::=	*Arith-expr* {`concat-int` *Arith-expr*}*
Arith-expr	::=	*Arith-term* {*Additive-op* *Arith-term*}*
Arith-term	::=	*Arith-factor* {*Multiplicative-op* *Arith-factor*}*
Arith-factor	::=	*Number*
	::=	(*Concat-expr*)
Additive-op	::=	+ | -
Multiplicative-op	::=	* | /

In this case, you should *not* assume that input is in scheme list form. The input will be actual source code; that is, a string of characters. Use SLLGEN (see Appendix B in the book).

You will first need to parse the expression, then interpret the syntax tree produced by the parser to evaluate it as an arithmetic expression. The parser takes care of the usual arithmetic precedence operations as well as precedence for `concat-int`, but the interpreter will have to take care of associativity, that is, making sure that operations at the same precedence level (e.g. additions and subtractions) are performed from left to right. Observe that there are no variables in these expressions. This interpreter need not take an environment parameter.

Name the interpreter `infix-interpreter`. Example output:

```
> (infix-intepreter "1 + 2 * 3 * (1 + 5)")
37
> (infix-interpreter "10 concat-int 24")
1024
> (infix-interpreter "1 + (23 concat-int 10)")
2311
> (infix-interpreter "1 + 23 concat-int 10")
2410
```

3. Implement an interpreter for the following language (which extends the language we did in the class with a switch statement):

$$\begin{array}{lcl} \mathit{Expression} & ::= & \mathit{Number} \\ & | & \mathit{Variable} \\ & | & -\ (\mathit{Expression}\ ,\ \mathit{Expression}\ ) \\ & | & \texttt{let}\ \mathit{Variable} = \mathit{Expression}\ \texttt{in}\ \mathit{Expression} \\ & | & \texttt{switch}\ \mathit{Expression}\ (\ \{\ (\ \texttt{case}\ \mathit{Expression}\ \texttt{->}\ \mathit{Expression}\ )\ \}^*\ ) \end{array}$$

Note that the comma in the second production above is part of the language. A switch expression works as you might expect:

```
> (case-interpreter "let x = 5 in
                     switch x (
                       (case -(3,2) -> 7)
                       (case -(7, -(x, 3)) -> 15))")

15

> (case-interpreter "let x = 5 in
                     switch -(x,3) (
                       (case -(4,2) -> 7)
                       (case -(4, -(x, 3)) -> 15))")

7

> (case-interpreter "let x = 5 in
                     switch x (
                       (case -(3,2) -> 7)
                       (case -(4, -(x, 3)) -> 15))")

'undefined
```

Note that the case expressions in a switch are evaluated top to bottom, and the first match is used. You should assume the input program is given as a string of characters, and use the SLLGEN system (see Appendix B of the book).