UNIVERSITY OF OSLO
Department of Informatics

A LISP Interpreter in ML

Mandatory Assignment 1

INF3110

September 21, 2009

Contents

1 Introduction

The purpose of this assignment is to write an interpreter, in ML, for a subset of the LISP dialect Scheme. An interpreter is a program that executes instructions written in a programming language. Large parts of the interpreter are already completed in the pre-code, which is posted at the course homepage.

LISP is chosen as the target language for four reasons:

1. Ease of scanning and parsing (not curriculum, completed in pre-code)
2. The dynamic nature of the language.
3. Its compactness and expressiveness.
4. Small core, most of LISP is written in LISP.

2 General Description of the Interpreter

The interpreter accepts input, from keyboard or from a file. A part of the program called a lexer (given as pre-code) converts a stream of characters to a stream of tokens. Some of these tokens has meta-data. An ID token has a name as meta-data, while a LPAR (left-parenthesis) token has none.

The parser will try to make sense of the stream of tokens delivered from the lexer and build a data structure, called a parse tree. For example, if a LPAR token (left-parenthesis) is read, the parser creates a List containing every structure until the enclosing RPAR token is read. The parse tree is constructed using different variants of the sexp datatype.

The sexp data type represents both code and data internally in the interpreter. Most procedures accept and return values of this datatype. Remember that pattern matching in ML may be used to expose the encapsulating data and to act on the different sexp variants in a meaningful way. Here's the definition of the sexp datatype used in the pre-code.

```
datatype sexp = Int of int
              | Symbol of sexp
              | Name of string
              | List of sexp list
              | Procedure of sexp WordHashTable.
                 hash_table list
                             * sexp list
                             * sexp list option
              | Bool of bool
              | VOID
              | DOT;
```

The environment is implemented as a list of S-expression Hash Tables, where each hash table corresponds to a given scope level. The base scope consists of all the primitives in the target language. The three procedures *scopeLookup*, *scopeAdd* and *scopeCreate* are recommended for maintaining the environment. The hash table is a mutable data structure.

Eval evaluates S-expressions in relation to an environment. A S-expression typically evaluates to another (simpler) S-expression, or to itself. A list S-expression is evaluated as a procedure call. This is a quite complex operation and should be performed with the help of the three procedures: *procedureCall*, *primitive* and *bindParamsToValues*.

The Procedure constructor in the sexp datatype is defined as:

```
Procedure of sexp WordHashTable.hash_table list * sexp
    list * sexp list option
```

This tells us that a procedure consists of an environment, a list of parameter names and an optional body of a list of S-expressions. If the body is NONE (the body is optional), then the procedure is a primitive and the implementation must be done in ML. Otherwise, the body is SOME S-expression that may be evaluated, together with an environment, as the procedure body.

3 What has to be Implemented

The following procedures must be implemented:

- stringRep
- primitive
- procedureCall (done for primitives)
- bindParamsToValues

3.1 stringRep

This procedure should accept any form of a S-expression and return a meaningful string representation. For example, the sexp List(Int 1, Int 2, Int 3) should have the string representation of "(1 2 3)".

3.2 primitive

This procedure should have a match against every target procedure primitive. All supported primitives are listed in the variable named primitives (see precode). The arguments to the procedure are not evaluated before entry. This is vital for, among other, the if statement. The if-statement should only evaluate one of the two possible bodies, depending on the if-test result. The usage of each primitive in Scheme is described in section **??**.

3.3 procedureCall

Normal procedures must get a local scope before their evaluation. It's also vital to make sure that the parameters are correctly initialized (see bindParamsToValues).

3.4 bindParamsToValues

Receives an environment, a list of sexp Names and a list of argument values. Performs the mapping between these lists. Make sure to support the sexp DOT notation used to express an unknown number of arguments that should be bundled together as a list.

procedure that counts the number of arguments

```
(define (param-count . l)
        (length l))

(param-count ) ; 0
(param-count 3) ; 1
(param-count 3 3 3) ; 3
```

4 A Crash Course in Scheme

The target language is a subset of Scheme that supports integers, lists, symbols and closures. It's missing strings, floating point numbers, tail call optimization, mutable state and some other advanced features of the language.

4.1 Lists

A list is something encapsulated by parentheses. This may be numbers, names (+, -, map, etc), symbols (something quoted), procedures, or other lists.

a list

```
(+ 1 2 3)
```

4.1.1 Procedure Calls

When the interpreter evaluates lists, it treats them as procedure calls. The first element of the list is expected to be a procedure that operates on the rest of the elements.

Adds the numbers 3 5 7

```
(+ 3 5 7) ; -> 15
```

4.1.2 Precedence

There's no operator precedence rules, since the use of parenthesis removes any ambiguity.

No operator precedence

```
(* (+ 1 2) 3)
```

4.1.3 Quote

A quote delays the evaluation of the following item. So, when something quoted is evaluated, it evaluates to itself. In particular, a delayed list will evaluate to a list. It's usual to refer to a delayed list as a list and a normal list as a procedure call. This is because of it's usage. When we write an interpreter, we need to focus on the correct definition.

A list of four elements

```
'(+ 3 5 7) ; -> (+ 3 5 7)
```

A symbol

```
'+ ; -> +
```

4.2 Print

Takes one argument and outputs it.

print

```
(print '(1 2 3))
(1 2 3)
```

4.3 Variable and Procedure Definitions

Since procedures are first-class values, variables and procedures are defined in a similar manner.

two variables

```
(define a 3)
(define b (+ a 3))
```

a procedure

```
(define (c d e)
(+ d e))
```

procedure call

```
(c 3 2) ; --> 5
```

procedure call with variable form earlier example

```
(c a b) ; --> 9
```

Procedures with an unknown number of arguments are defined using a dot notation.

a procedure with an unknown number of arguments

```
(define (sum . l)
(apply + l))
```

procedure call

```
(sum 1 2 3 4 5 6 7 8 9 10) ; --> 55
```

4.4 Lambda Procedures

The lambda primitive is used to create anonymous procedures. They may also be bound to a name using *let* or *define*.

lambda compared to regular procedure definitions

```
(define a1 (lambda (x y) (+ xy))) ; proc named a1 adds
    its two arguments
(define (a2 x y) (+ xy)) ; proc named a2 adds its two
   arguments
```

Lambda is often used when a small code snippet is needed.

lambda usage

```
(map (lambda (x) (+ x 1))
     '(1 2 3 4)) ; -> '(2 3 4 5)
```

4.5 Let

Let is used to create local variable bindings.

let usage

```
(define (complexProc a b c)
    (let ((i (* a b))
          (k (* b c)))
        (+ i k)))
```

4.6 List operations: cons, car and cdr

The last element of every list is the empty list.

The empty list

```
'()
```

This is for all our purposes exactly the same as nil in ML and null in Java.

4.6.1 Cons

Another way to create a list is to use the procedure cons. Cons takes an element and a list as arguments and creates a new list with the element argument in front of the the list argument. NB: our implementation allows a cons of two elements to become a new list of two elements. This makes the implementation a bit easier. This would become a pair of two elements in ordinary LISP.

normal cons

```
(cons 1 '(2 3)) ; --> '(1 2 3)
```

normal canonical cons

```
(cons 1 (cons 2 '())) ; --> '(1 2)
```

non canonical cons

```
(cons 1 2) ; --> '(1 2)
```

4.6.2 Car

Car returns the first element in a list

car of a list

```
(car '(1 2 3)) ; --> 1
```

4.6.3 Cdr

Cdr returns a list without the first element.

cdr of a list

```
(cdr '(1 2 3)) ; --> '(2 3)
```

4.6.4 List procedure

Takes an unknown number of arguments and constructs a list of them.

the list procedure

```
(list 1 2 3); -> '(1 2 3)
(list 1 (+ 2 3) 4) -> '(1 5 4)
```

4.7 Arithmetic

The interpreter only supports whole numbers at this point.

4.7.1 Addition

some addition

```
(+ 1 2 3) ; -> 6
(+) ; -> 0
(+ 1) ; -> 1
```

4.7.2 Subtraction

some subtraction

```
(- 10 2 3) ; -> 5
(- 1) ; -> ~1
(-) ; -> error
```

4.8 Multiplication

usage of *

```
(* ) ; -> 1
(* 1 2 3) ; -> 6
```

4.9 Remainder

The leftover after an integer division

```
(remainder 3 7) ; -> 3
(remainder 13 7) ; -> 6
```

4.10 Quotient

The result of an integer division

```
(quotient 3 7) ; -> 0
(quotient 13 7) ; -> 1
```

4.11 Comparisons

4.11.1 Null?

Null? checks if a list is empty

null? tests

```
(null? '()) ; -> true
(null? '(1)) ; -> false
```

4.11.2 Pair? and List?

These are identical in our implementation (pair? is a primitive and list? is implemented using pair? in the file stdlib.scm)

pair? tests

```
(pair? '(1)) ; -> true (contains pair of 1 and '() )
(pair? '()) ; -> false
(pair? 1) ; -> false
```

4.11.3 Equal

equality test

```
(= 1 2 3) ; false
(= 1 1 1 (- 3 2)) ; true
```

4.11.4 Greater Than

greater than

```
(> 3 2 1) ; true
(> 3 2 2)  ; false
(> 5 2)   ; true
```

4.11.5 Less Than

less than

```
(< 1 2 3) ; true
(< 1 2 2)  ; false
(< 2 5)   ; true
```

4.12 What's true?

In LISP, it's more correct to ask what's false. False is false, everything else is true.

4.12.1 And

Evaluates an unknown number of arguments and checks that none is false. Returns the last value if all true, else false.

usage of and

```
(and ) ; -> true
(and (= 3 2) (= 2 2)) ; -> false
(and 1 2 3) ; 3
```

4.12.2 Or

Evaluates an unknown number of arguments and checks if any of them is true.

usage of or

```
(or ) ; -> false
(or (= 3 2) (= 2 2)) ; -> true
(or 1 2 3) ; -> 1
```

Calling eval from lisp There's a ML procedure called eval that tries to evaluate every lisp data structure. This procedure is the heart of the interpreter and we're also able to call it directly from lisp. There's one subtle difference:

the eval procedure is special

```
call.
'(+ 1 2 3) ; -> (+ 1 2 3)
(eval '(+ 1 2 3)) ; -> 6
```

The ability to dynamically create and evaluate data is a very useful feature of LISP.

5 Programming Tips

ML has a strict type system and some quite obscure error messages. This page tries to explain the different messages [http://www.smlnj.org/doc/errors.html] .

The SML/NJ compiler does not support a debugger, or even a stack trace (backtrace). It's therefore very useful to develop each procedure locally and test them using constructed arguments. The *ptest()* procedure will return a parsed result of its input, which might come in handy.

Emacs has a nice SML-mode that makes developing a bit easier.

- `http://www.smlnj.org/doc/Emacs/sml-mode.html`

Send a buffer to the compiler using 'C-c C-b' and jump to the nearest error using 'C-x ‘'.

There are a couple of books about Standard ML at the library, the website [http://www.smlnj.org/] contains some documentation and tutorials.

If you're unsure about how your interpreter should respond in a given situation, try a real Scheme interpreter. DrScheme should be installed on all Windows and Linux machines at IFI.

Some free books on Scheme:

- `http://www-mitpress.mit.edu/sicp/full-text/book/book.html`
- `http://t3x.org/sketchy/vol1/index.html`
- `http://www.ccs.neu.edu/home/dorai/t-y-scheme/t-y-scheme.html`

5.1 Final Test

As a final test of the interpreter, try to uncomment the line loading the file 'interpreter-tests.scm'. This will make the interpreter run a few test programs at startup. Passing these tests is a good indication of a working interpreter.