CMSC 22600 **Autumn 2016**	**Compilers for Computer Languages**	**Homework 1** **Due Friday Oct 7**

Overview

For this assignment, you will implement a small Scheme-like language called μSCHEME. The project consists of two main pieces: a parser that translates input into a tree representation and an interpreter that executes the program. The main purpose of this assignment is to get your feet wet with SML programming and to explore a simple example of end-to-end language implementation.

This rest of this document is organized into four parts: the specification of μSCHEME, some examples of μSCHEME programs, a discussion of how to implement the specification, and information about submitting your work.

μSCHEME

This section describes the concrete syntax and dynamic semantics of μSCHEME.

Syntax

As is standard, we split the discussion of μSCHEME's syntax into lexical issues (*i.e.*, how the input text is organized into tokens) and the syntactic structure of the tokens.

Lexical issues

μSCHEME programs are written using a subset of the ASCII (7-bit) character set. The sequence of characters in a program are logically grouped into *tokens* (*e.g.*, punctuation, identifiers, numbers, *etc.*). Whitespace (*i.e.*, sequences of space, tab, carrage return, and newline characters) may be used to separate tokens.[1] In addition to whitespace between tokens, a μSCHEME program may also have *comments*, which consist of uninterpreted text beginning with the semicolon character ('`;`') and ending with the next newline character.

μSCHEME has two punctuation symbols: left ('`(`') and right ('`)`') parentheses. In addition, it has two classes of terminal symbols, which are specified as follows:

$$\begin{aligned} \mathit{id} &::= \mathit{letter}^{+} \\ \mathit{num} &::= \texttt{-}^{?}\mathit{digit}^{+} \end{aligned}$$

[1] In some cases, such as two identifier tokens in sequence, it is necessary to have separating whitespace.

Here the notation '*letter*$^+$' means *one or more letters in sequence*, and '**-**$^?$' means *zero or one occurrences of the character* '**-**'. Two identifiers, '**define**' and '**lambda**', are *reserved words* (*a.k.a. keywords*), which have special status in the syntax and cannot be used as variables.

μSCHEME's grammar

A μSCHEME program consists of a sequence of zero or more definitions followed by an expression. Expressions are either variables, numbers, applications, or λ abstractions. The following context free grammar (CFG) specifies the syntax of μSCHEME programs:

$$
\begin{array}{rcl}
\mathit{prog} & ::= & \texttt{(define } \mathit{id}\ \mathit{exp}\ \texttt{)}\ \mathit{prog} \\
& \mid & \mathit{exp} \\
\\
\mathit{exp} & ::= & \mathit{id} \\
& \mid & \mathit{num} \\
& \mid & \texttt{(}\ \mathit{exp}^+\ \texttt{)} \\
& \mid & \texttt{(lambda (}\ \mathit{id}^*\ \texttt{)}\ \mathit{exp}\ \texttt{)}
\end{array}
$$

Note that the grammar is defined in terms of the language's tokens; we do not mention whitespace or comments, since that would just add noise to the grammar, but whitespace and/or comments may occur between any two symbols in the grammar.

Dynamic semantics

We can specify the execution behavior of μSCHEME programs with a simple operational semantics. You can think of such a semantics as an abstract description of an interpreter. Let us first define some semantic domains with the following equations:

$$
\begin{array}{rcll}
\rho \in \text{ENV} & = & \text{IDENT} \overset{\text{fin}}{\to} \text{VALUE} & \text{Environments} \\
v \in \text{VALUE} & = & \mathbb{Z} \cup \text{CLOS} & \text{Values} \\
\langle \Lambda, \rho \rangle \in \text{CLOS} & = & \text{LAMBDA} \times \text{ENV} & \text{Closures} \\
n \in \mathbb{Z} & = & \{\ldots, -2, -1, 0, 1, 2, \ldots\} & \text{Integers} \\
\Lambda \in \text{LAMBDA} & & & \text{Lambda expressions}
\end{array}
$$

Environments are finite maps from identifiers to runtime values, values are either integers or function closures, and closures are a pair of a lambda expression and an environment that defines the free variables of the expression.

The dynamic semantics of μSCHEME is given by the relation $\rho \vdash e \Downarrow v$, which can be read as saying that given an environment ρ, the expression e evaluates to the value v. Formally speaking, this relation is defined as the *least* relation satisfying the rules given in Figure 1.

The μSCHEME basis

We use the term *basis* to describe the initial (or predefined) environment in which a μSCHEME program executes. The basis maps identifiers to builtin operations that cannot be directly defined by the above semantics. For μSCHEME, we define the following basis functions:

$$[\textsc{Abs}]\ \frac{}{\rho \vdash \Lambda \Downarrow \langle \Lambda,\ \rho \rangle} \qquad [\textsc{Ident}]\ \frac{x \in \mathrm{dom}(\rho)}{\rho \vdash x \Downarrow \rho(x)} \qquad [\textsc{Num}]\ \frac{}{\rho \vdash n \Downarrow n}$$

$$[\textsc{Apply}]\ \frac{\begin{array}{c}\rho \vdash e_0 \Downarrow \langle\ \texttt{(lambda (}x_1 \cdots x_n\texttt{)}\ e\texttt{)},\ \rho' \rangle \\ \rho \vdash e_1 \Downarrow v_1 \qquad \cdots \qquad \rho \vdash e_n \Downarrow v_n \\ \rho'[x_1 \mapsto v_1,\ \ldots,\ x_n \mapsto v_n] \vdash e \Downarrow v\end{array}}{\rho \vdash (e_0\ e_1\ \cdots\ e_n) \Downarrow v}$$

$$[\textsc{Def}]\ \frac{\rho \vdash e \Downarrow v \qquad \rho[x \mapsto v] \vdash p \Downarrow v'}{\rho \vdash \texttt{(define}\ x\ e\texttt{)}\ p \Downarrow v'}$$

Figure 1: Dynamic semantics for μSCHEME

- `(add` $n_1 \cdots n_k$`)` *where* $0 \leq k$ — integer addition.
- `(mul` $n_1 \cdots n_k$`)` *where* $0 \leq k$ — integer multiplication.
- `(equal` n_1 n_2`)` — integer equality.
- `(if` n v_1 v_2`)` — conditional.

The semantics of these operations is given by the $\delta[\![\cdot]\!]$ function, which is defined as follows:

$$\begin{array}{rcl}
\delta[\![\texttt{(add)}]\!] & \Rightarrow & \texttt{0} \\
\delta[\![\texttt{(add}\ n\ \cdots\texttt{)}]\!] & \Rightarrow & n + \delta[\![\texttt{(add}\ \cdots\texttt{)}]\!] \\
\delta[\![\texttt{(mul)}]\!] & \Rightarrow & \texttt{1} \\
\delta[\![\texttt{(mul}\ n\ \cdots\texttt{)}]\!] & \Rightarrow & n * \delta[\![\texttt{(mul}\ \cdots\texttt{)}]\!] \\
\delta[\![\texttt{(equal}\ n\ n\texttt{)}]\!] & \Rightarrow & 1 \\
\delta[\![\texttt{(equal}\ n_1\ n_2\texttt{)}]\!] & \Rightarrow & 0 \quad \textit{where}\ n_1 \neq n_2 \\
\delta[\![\texttt{(if}\ n\ v_1\ v_2\texttt{)}]\!] & \Rightarrow & v_1 \quad \textit{if}\ n \neq 0 \\
\delta[\![\texttt{(if}\ 0\ v_1\ v_2\texttt{)}]\!] & \Rightarrow & v_2
\end{array}$$

Examples

In this section, we present a few simple examples of μSCHEME programs.

We can define a subtraction function using addition and multiplication:

```
(define sub (lambda (a b) (add a (mul -1 b))))

(sub 5 7)
```

We typically do not want to evaluate both arms of a conditional, so we can enclose them in λ abstractions and then apply the result of the conditional by adding an extra set of enclosing parentheses:

```
((if (equal 1 2) (lambda () 3) (lambda () 4)))
```

While μSCHEME does not have recursion, we can define a *fixed-point* combinator to implement recursion:

```
; fixed-point combinator
(define fix
  (lambda (f) ((lambda (x) (f (lambda (v) ((x x) v))))
               (lambda (x) (f (lambda (v) ((x x) v)))))))

; recursive factorial function defined using fix
(define fact
  (fix
    (lambda (rfact)
     (lambda (n) ((if n
                      (lambda () (mul n (rfact (add n -1))))
                      (lambda () 1)))))))

(fact 5)
```

Implementation

Your assignment is to implement the specification from above. We will seed your phoenixforge repository with a directory called `hw1` that contains the following files:

- `hw1.cm` — the CM file for building your program
- `eval.sml` — the μSCHEME interpreter
- `main.sml` — the driver that connects the parser and interpreter together
- `parser.sml` — the μSCHEME parser
- `print.sml` — a printer for results
- `tree.sml` — a syntax-tree representation of μSCHEME programs

You will need to complete the code in `eval.sml` and `parser.sml`.

Program representation

Programs are represented as as syntax trees, which are directly encoded as the following SML datatypes in the `Tree` structure (`tree.sml`):

```
type id = Atom.atom

datatype prog
  = Define of id * exp * prog
  | Exp of exp

and exp
  = Var of id
  | Num of IntInf.int
  | Apply of exp * exp list
  | Lambda of id list * exp
```

The `Atom` module provides a representation of strings that support fast (constant-time) comparisons and hashing.

Parsing

In SML, a *reader* is a function that takes an input stream and returns either `NONE` (for errors or end-of-stream) or `SOME(v,s)`, where v is a value and s is the rest of the stream.

```
type ('a, 'strm) reader = 'strm -> ('a * 'strm) option
```

The `reader` type is defined in the `StringCvt` structure in the SML Basis Library.

We structure the μSCHEME parser into three levels: the first classifies characters by type:

```
datatype chr_cls
  = LP | RP | MINUS | SEMI  (* '(', ')', '-', and ';' *)
  | LETTER of char          (* 'A'..'Z' and 'a'..'z' *)
  | DIGIT of int            (* '0'..'9' *)
  | WS                      (* white space characters (other than '\n') *)
  | EOL                     (* end-of-line '\n' *)
  | OTHER                   (* any other character *)
```

We provide a function that given a character reader will return a `chr_cls` reader.

```
val getcc : (char, 'strm) CvtString.reader
              -> (chr_cls, 'strm) CvtString.reader
```

On top of this layer, we define a representation of tokens:

```
datatype token
  = LPAREN | RPAREN         (* '(' and ')' *)
  | LAMBDA                  (* 'lambda' *)
  | DEFINE                  (* 'define' *)
  | IDENT of Atom.atom      (* identifiers *)
  | INT of IntInf.int       (* integer literals *)
```

You should define a function that takes a character reader and returns a token reader:

```
val getToken : (char, 'strm) CvtString.reader
                -> (token, 'strm) CvtString.reader
```

Your implementation of this function should use the character-classifier that we provide.

Lastly, you will implement a parsing function that takes a character reader and a character stream as inputs and returns a program:

```
val parseProg : (char, 'strm) CvtString.reader -> 'strm -> Tree.prog
```

as part of your implementation you will need to implement a function for parsing expressions (either nested inside `parseProg` or defined at top level). For this assignment, we will implement the simplest form of error handling. Namely, we will raise a `Fail` exception with a useful message when we encounter a syntactic error in the input.

To parse the input, you will use a technique called *recursive decent*. Essentially, for each nonterminal in the grammar (*e.g.*, *exp*), there will be a function (*e.g.*, `parseExp`) that takes the input stream as an argument and returns the result of the parse and the remaining input. For example, if the input was the string

```
(add 1 (mul 2 3)) ...
```

Then calling `parseExp` would return the syntax tree

```
Apply(Var(Atom.atom "add"), [
  Num 1,
  Apply(Var(Atom.atom "mul"), [
    Num 2,
    Num 3
  ])])
```

along with the remaining input (*i.e.*, "`...`"). The `parseExp` function looks at the next token in the input stream and then makes a decision about what it expects to parse.[2]

```
(* parse an expression *)
  fun parseExp strm = (case getTok strm
         of NONE => raise Fail "error: unexpected end of file"
          | SOME(LPAREN, rest) =>
              ... (* parse application or lambda abstraction *) ...
          | SOME(LAMBDA, rest) => raise Fail "error: unexpected 'lambda'"
          | SOME(DEFINE, rest) => raise Fail "error: unexpected 'define'"
          | SOME(RPAREN, rest) => NONE
          | SOME(IDENT x, rest) => SOME(Tree.Var x, rest)
          | SOME(INT n, rest) => SOME(Tree.Num n, rest)
        (* end case *))
```

Here we are assuming that the `getTok` function is defined by applying `getToken` to the character reader supplied to `parseProg`.

You will probably want to define additional functions for parsing applications and λ abstractions, *etc.*.

2 In the case of an application or λ abstraction, it must look at two tokens.

Evaluation

Once a program is parsed it can be evaluated. The `eval` function (defined in the `Eval` structure) has the type

```
val eval : env * Tree.prog -> value
```

where the `env` type is defined to be

```
type env = value AtomMap.map
```

(*i.e.*, a finite map from identifiers to values).

Runtime values are represented by the `value` type, which has the following definition in the `Eval` structure (`eval.sml`):

```
datatype value
  = NUM of IntInf.int
  | CLOS of value list -> value
```

Note that we represent closures as SML functions. This representation allows us to handle builtin functions using the same representation as for user-define λ abstractions For example, we can implement the n-ary `add` function as

```
fun add args = let
      fun add' (NUM n, s) = (n + s)
        | add' (CLOS _, _) = raise Fail "add␣expects␣numbers,␣given␣closure"
      in
        NUM(List.foldl add' 0 args)
      end
```

and define an initial environment that maps predefined function names to the builtin functions

```
val basis = let
      fun ins ((name, f), env) =
            AtomMap.insert (env, Atom.atom name, CLOS f)
      in
        List.foldl ins AtomMap.empty [
            ("add", add),
            ("mul", mul),
            ...
          ]
      end
```

Submission

We will seed your pheonixforge svn repository with a collection of source-code files that comprise this project in a directory called `hw1`. You can check out a copy of your repository using the svn command:

```
svn co https://phoenixforge.cs.uchicago.edu/svn/CNETID-cs226-aut-16 cs226
```

where `CNETID` is your University of Chicago login. Two of the files in the repository (`hw1/parse.sml` and `hw1/eval.sml`) contain unimplemented functions that you will need to complete. Please submit your solution by committing your changes to your pheonixforge svn repository. The assignment is due at **11pm on Friday October 7**.

History

2016-10-01 Fixed mismatched parentheses in AST example.

2016-09-28 Added repository URL.

2016-09-27 First version.