Scheme Overview

This handout is a brief summary of the **subset** of the Scheme programming language presented in lecture on Feb. 10-12. Refer to the Dr. Scheme documentation and "Revised[6] Report on the Algorithmic Language Scheme" at r6rs.org for detailed reference material. The bulk of this handout is a casual approach intended to help you complete the homework assignment, although I include a short formal specification of Scheme expressions we're using in class at the end.

Scheme has an abstract syntax where the parse tree is directly written using nested parentheses. The first identifier after a parenthesis in an expression generally denotes the kind of expression. This syntax makes it very easy for a machine to parse and manipulate Scheme code (if not for a human!). There are a number of variations of Scheme; we're primarily using the PLAI variant in class.

Parentheses in Scheme are the characters ( ) [ ] { }. Each pair is interchangeable; they are varied solely for readability. An **expression** (also called symbolic expression, s-expression, s-expr, and sexp) is a sequence of literals and identifiers. The expressions that we consider are literals, variables, IF, AND, OR, COND, LET, LETREC, LAMBDA, application. I write the names of expressions in capital letters when referring to them in text (to distinguish them from the same English words). Beware that Scheme is case sensitive and expects them in lowercase when actually typed.

Single-line comments in Scheme begin with a semi-colon.

1 Literals

Number literals include those from Java, e.g.,

```
1
-32
46.16
3e6
```

The **boolean** literals are

```
#t
#f
```

although you can also use true and false in some situations (but don't).

String literals are written in double-quotes, just like Java:

```
“hello world”
```

Symbols are a kind of immutable string with few functions that operate on them. Their value will become clear later; for now all that is important is that symbol literals can be created with a single quote (and no following quote).

```
'x
'hello-world
```

This is actually a shorthand for the **quote** special form, which looks like:

```
(quote x)
(quote hello-world)
```

A literal linked list can be created by quoting a series of literals surrounded by parentheses, e.g.,

```
' (1 2 3 4 5)
(quote 1 2 3 4 5)
'(1 a 73.2 Williams)
```

These can also be nested to form trees of lists:

```
'(1 (2 3) a b (c d + (x)))
```

The empty list is therefore:

```
'()
```

2 Identifiers

Identifiers (variable names) in Scheme may contain letters, numbers, and other characters except for period, parentheses, quotation marks, hash (#), and whitespace. They are case-sensitive and may not begin with a number. Some legal identifiers are:

```
factorial
a-long-function-name
sort!
+
boolean?
```

By convention, identifiers are usually lowercase and use minus signs to separate words, identifiers ending in question marks are used for predicate procedures that return a boolean value, and identifiers ending in exclamation marks (which is pronounced “bang”) denote procedures that mutate their arguments.

3 Forms

3.1 Conditionals

The value of the **if expression** is either the value of its consequent or alternate expression, depending on whether the test expression evaluates to true or false:

```
(if test-exp consequent-exp alternate-exp)
```

The test expression is always evaluated first and only one of the consequent or alternate is evaluated. Note that IF "returns" a value in Scheme, like the ? operator in Java.

Nested IF expressions can be collapsed into a single **cond expression** of the form:

```
(cond [test0-exp  consequent0-exp]
      [test1-exp  consequent1-exp] ...)
```

This evaluates each test expression in turn. The value of the case expression is the value of the consequent expression corresponding to the first test that evaluates to true. It is an error if none of the expressions evaluate to true. The special expression "else" is equivalent to using "true" as a test expression. Note that only one of the consequents is ever evaluated.

Logical comparisons >, <, >=, <=, equals? are implemented as procedures. The logical AND and OR operations are implemented as special forms because, like IF, they only evaluate a subset of their arguments (sometimes).

```
(and arg0 arg1 ...)
(or arg0 arg1 ...)
```

AND and OR each accept two or more arguments. They evaluate the first expression. For AND, if the first expression evaluates to false the value of the AND is false and the remaining arguments are not evaluated. Otherwise the value of the AND is the AND of the remaining values (i.e., each is evaluated in turn until one that is false is found). This is equivalent to the semantics of Java's && operator. OR is equivalent to Java's || operator; it evaluates each argument until one that is true is found or the end is reached.

3.2 Procedure Definition

Unlike Java, scheme separates the creation of a procedure from binding it to a name. This means that you can create anonymous (unnamed) procedures. The **LAMBDA expression** creates a procedure:

```
(lambda (id0 id1 ...) body-exp)
```

Evaluating a LAMBDA expression produces a procedure. It does not execute the procedure. There may be zero or more identifiers; those are the formal parameters. The body of a procedure must be a single expression, which is the value that it "returns" when applied. Example of lambda expressions are:

```
(lambda (x) (+ x 1))
(lambda (a b c) (if (> a b) c b))
```

Dr. Scheme allows you to insert the λ character directly into your program and use it instead of typing "lambda". I will also often write λ on the board as a shorthand.

3.3 Procedure Application

An **application expression** contains one or more expressions, the first of which must evaluate to a procedure:

```
(proc-exp arg-exp0 arg-exp1 …)
```

The value of an application expression is the return value of the procedure applied to those arguments. The order of evaluation of arguments in Scheme is undefined.

3.4 Binding (let)

The value of **LET expression** is the evaluation of a body expression in a new environment where the specified variables are bound to the evaluation of corresponding expressions. Its syntax is:

```
(let
     ([id0 exp0]
      [id1 exp1] …)
  body-exp
)
```

Example:

```
(let ([x (+ 1 0)]
      [y 2])
     (+ x y))
```

evaluates to 3. The expressions are evaluated in the environment outside the let expression. That is,

```
(let ([x 3])
        (let ([x 4]
              [y x])
             y))
```

Evaluates to 3.

The **LETREC expression** has the same structure as let, but evaluates the expressions in the new environment, which not only lets them refer to each other but allows recursive definitions, e.g.,

```
(letrec
     ([factorial  (lambda (x)
                      (if (> x 0)
                        (* x (factorial (- x 1)))
                        1)) ])
  (factorial 4))
```

evaluates to 24.

There's another binding expression called **define**. It is incredibly convenient, but theoretically inelegant. It has two forms:

(define *identifier expression*)

Creates a new identifier in the current environment and binds it to the value of the expression. This is sort-of (but not actually) like having a LET that wraps the entire rest of the program. The other syntax,

(define (*proc-identifier id0 id1 …*) *body-exp*)

creates a procedure and binds it. It is semantically equivalent to:

(define *proc-identifier* (lambda (*id0 id1 …*) *body-exp*))

4 Some Procedures

(list *a b c …*)
Creates a linked list of the (zero or more) arguments.

(cons *a L*)
Creates a new linked list node with *a* as the first value and *L* as the rest of the list.

(first *L*)
(car *L*)
Returns the first element of linked list *L*.

(rest *L*)
(cdr *L*)
Returns the second node of linked list *L*.

(second *L*)
(cadr *L*)
Returns the second *element* of linked list *L*.

(+ *a b* …)
Returns the sum of the (two or more) arguments. -, /, *, etc. all work as you might expect; see the reference documentation.

(not *b*)
Returns the logical negation of boolean value *b*.

(eq? *a b*)
Returns true if *a* and *b* are semantically pointers to the same value (like Java's ==)

(equal? *a b*)
Returns true if *a* and *b* have equivalent values (like Java's equal method)

(procedure? *a*)
(boolean? *a*)
(number? *a*)
(symbol? *a*)
(string? *a*)
(list? *a*)
Each returns true if the argument expression has the appropriate type.

(null? *a*)
True if *a* is eq? to '()

(odd? *a*)
(even? *a*)
True if the number is odd (or even).

(append *L0 L1* ...)
Produce a new list that contains the elements of one or more lists that are the arguments. The last list is used as the tail of the resulting list (i.e., it is shared).

(map *proc-exp L0* ...)
Applies a procedure each element to one or more lists, producing a new list of the results.

5 Scheme Expression Domain BNF

The true Scheme BNF is given in R^{6}RS. This is a subset, and I've simplified the number hierarchy and production grammar. For readability, I highlighted terminals instead of enclosing them in quotation marks. I've omitted the specification of where whitespace can occur; this introduces ambiguity but is helpful for most human readers. As is common practice, I'm using some regular expression shorthand to avoid introducing lots of helper productions for the cond and let clauses. The shorthand uses the following conventions:

[x] = zero or one x's
(x) = same as x; just used for grouping
x* = zero or more x's
x$^+$ = one or more x's

literals:

```
<bool>           ::= #t | #f
<id>             ::= <letter> (<letter> | <digit>)*
<symbol>         ::= ' <id>
<digit>          ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<int>            ::= [ - | + ] <digit>+
<decimal>        ::= <int> .  <digit>+
<num>            ::= <int> | <decimal>
<list-lit>       ::= <id> | <num> | <bool> |  ( <list-lit>* )
<list>           ::= '  ( <list-lit>* )
<literal>        ::= <bool> | <num> | <symbol> | <list>
```

expressions:

```
<define-simple>  ::= ( define <id>  <exp> )
<define-lambda> ::= ( define ( <id>+ )  <exp> )
<define>         ::= <define-simple> | <define-lambda>
<variable>       ::= <id> that is not a reserved word
<application>    ::= ( <exp>+ )
<if>             ::= ( if  <exp> <exp> <exp> )
<let>            ::= ( let  ( ( ( <id> <exp> ) )* ) <exp> )
<letrec>         ::= ( letrec ( ( ( <id> <exp> ) )* ) <exp> )
<cond>           ::= ( cond  ( (<exp> <exp> ) )* [ ( else <exp> )  ] )
<lambda>         ::= ( lambda ( <id>* )  <exp> )

<exp>            ::=       <literal> |
                           <variable> |
                           <application> |
                           <if> |
                           <let> |
                           <letrec> |
                           <cond> |
                           <define> |
                           <lambda>
```

("letter" in Scheme includes all ASCII characters except digits, parentheses, semi-colon, whitespace and period.)

6 Scheme Value Domain

The Scheme value domain (that we've studied so far...) contains integers, reals, booleans, symbols, lists, and procedures. It can be expressed as a gammar simply by:

```
<value> ::= <literal> | <proc>
```

where <proc> is the type of an evaluated LAMBDA expression. Beware that we're being a little casual here with abstractions. From a software engineering perspective, there should be a stronger difference between a literal expression and the value that comes from it. However, for the purpose of creating a mathematical model of evaluation by "substitution" and a very straightforward interpreter, it is convenient to treat those as the same.

Value domains can also be expressed in set notation. The Scheme value domain that we've studied so far looks like:

symbol = { 'a 'b 'c ... }
$\text{number} = \text{integer} \cup \text{real}$

$\text{simple-value} = \text{symbol} \cup \text{number} \cup \text{boolean}$

$\text{list}_0 = \{\text{empty-list}\}$
$\text{list}_i = \{\text{simple-value}\} \times \text{list}_{i-1}$
$\text{list} = \bigcup_i \text{list}_i$

$\text{value} = \text{simple-value} \cup \text{procedure}$

The $\cup$ means union; put together the elements from two sets, for example, $\{1, 2\} \cup \{a, b\} = \{1, 2, a, b\}$. The big $\cup$ is like a summation for unions. In this case it means that "list" is the union of lists of length 1, lists of length 2, etc. up to infinitely long lists. Note that lists of varying lengths were defined by a process similar to induction, which also follows the long way (i.e., without regular expressions) of defining integers and lists in the grammar. The $\times$ means product. In the context of sets, it defines a set that contains all combinations of one element from the left and one element from the right. For example, $\{1, 2\} \times \{a, b\} = \{1a, 1b, 2a, 2b\}$.