5.3. PROGRAMMING WITH SCHEME

Scheme is a multi-paradigm programming language. It is a dialect of Lisp which supports functional and procedural programming. It was developed by Guy L. Steele and Gerald Jay Sussman in the 1970s. Scheme was introduced to the academic world via a series of papers now referred to as Sussman and Steele's Lambda Papers. There are two standards that define the Scheme language: the official IEEE standard, and a de facto standard called the Revisedn Report on the Algorithmic Language Scheme, nearly always abbreviated RnRS, where n is the number of the revision. The current standard is R5RS, and R6RS is in development.

Scheme's philosophy is minimalist. Scheme provides as few primitive notions as possible, and, where practical, lets everything else be provided by programming libraries.

Scheme was the first dialect of Lisp to choose static (a.k.a. lexical) over dynamic variable scope. It was also one of the first programming languages to support first-class continuations.

- A mid-1970s dialect of LISP, designed to be cleaner, more modern, and simpler version than the contemporary dialects of LISP, Uses only static scoping

- Functions are first-class entities, They can be the values of expressions and elements of lists, They can be assigned to variables and passed as parameters

- Primitive Functions:

1. Arithmetic: +, -, *, /, ABS, SQRT

Ex: (+ 5 2) yields 7

2. QUOTE: -takes one parameter; returns the parameter without evaluation

- QUOTE is required because the Scheme interpreter, named EVAL, always evaluates

parameters to function applications before applying the function. QUOTE is used to avoid parameter evaluation when it is not appropriate

- QUOTE can be abbreviated with the apostrophe prefix operator

e.g., '(A B) is equivalent to (QUOTE (A B))

3. CAR takes a list parameter; returns the first element of that list

e.g., (CAR '(A B C)) yields A

(CAR '((A B) C D)) yields (A B)

4. CDR takes a list parameter; returns the list after removing its first element

e.g., (CDR '(A B C)) yields (B C)

(CDR '((A B) C D)) yields (C D)

5. CONS takes two parameters, the first of which can be either an atom or a list and the second of which is a list; returns a new list that includes the first parameter as its first element and the second parameter as the remainder of its result

e.g., (CONS 'A '(B C)) returns (A B C)

6. LIST - takes any number of parameters; returns a list with the parameters as elements

- Predicate Functions: (#T and () are true and false)

1. EQ? takes two symbolic parameters; it returns #T if both parameters are atoms and the two are the same

e.g.,(EQ? 'A 'A) yields #T

(EQ? 'A '(A B)) yields ()

Note that if EQ? is called with list parameters, the result is not reliable Also, EQ? does not work for numeric atoms

2. LIST? takes one parameter; it returns #T if the parameter is an list; otherwise ()

3. NULL? takes one parameter; it returns #T if the parameter is the empty list; otherwise ()

Note that NULL? returns #T if the parameter is ()

4. Numeric Predicate Functions

=, <>, >, <, >=, <=, EVEN?, ODD?, ZERO?

5. Output Utility Functions:

(DISPLAY expression)

(NEWLINE)

- Lambda Expressions

- Form is based on notation

e.g.,

(LAMBDA (L) (CAR (CAR L))) L is called a bound variable

- Lambda expressions can be applied

e.g.,

((LAMBDA (L) (CAR (CAR L))) '((A B) C D))

- A Function for Constructing Functions

DEFINE - Two forms:

1. To bind a symbol to an expression

EX:

(DEFINE pi 3.141593)

(DEFINE two_pi (* 2 pi))

2. To bind names to lambda expressions

EX:

(DEFINE (cube x) (* x x x))

- Example use:

(cube 4)

- Evaluation process (for normal functions):

1. Parameters are evaluated, in no particular order

2. The values of the parameters are substituted into the function body

3. The function body is evaluated

4. The value of the last expression in the body is the value of the function

(Special forms use a different evaluation process)

Control Flow:

- 1. Selection- the special form, IF

(IF predicate then_exp else_exp)

e.g.,

(IF (<> count 0) (/ sum count) 0)

5.4 PROGRAMMING WITH ML

- A static-scoped functional language with syntax, that is closer to Pascal than to LISP

- Uses type declarations, but also does type inferencing to determine the types of undeclared variables

- It is strongly typed (whereas Scheme is essentially type less) and has no type coercions

- Includes exception handling and a module facility for implementing abstract data types

- Includes lists and list operations

- The val statement binds a name to a value (similar to DEFINE in Scheme)

- Function declaration form: fun function_name (formal_parameters) = function_body_expression;

e.g., fun cube (x : int) = x * x * x;

- Functions that use arithmetic or relational operators cannot be polymorphic--those with

only list operations can be polymorphic

Datatypes

ML has Constants of five types:

Integers -- a string of digits (0-9), optionally preceded by a tilde (~) to indicate negative

Reals -- string of digits (possibly preceded by ~), followed by at least one of a decimal point and one or more digits

the letter E, followed by an integer

Booleans -- true or false

Characters -- Ascii characters, e.g., #"A"

Strings -- a sequence of characters enclosed with double-quotes (")

\ is a special character, and means that the next character (or more) indicate something special

\n is the newline character, basically just like hitting the Return key

\t is the tab character -- like hitting the Tab key

\\ is the backslash -- since one \ is special, just type \\ to really mean backslash

\" is the double-quote -- a plain double-quote would mean the end of the string, to have a double-quote actually be in the string, you use \"

\### is a way to specify a character by its ASCII number

\^A for any letter A is a way to specify a control character

\ as the last character on the line means the string continues on the following line

- negation (unary minus) is evaluated first (highest precendence)
- multiplication operators are next
- addition operators are last
- parentheses, of course, override these rules (technically, they have highest precedence)

String operators:

^ is concatenation; that is, it makes one string out of two by slapping them together, end to end

"" is an empty string, and is allowed

Comparison (Relational) Operators: =, <, >, <=, >=, <>

- These can compare two values of the same type: integer, real, or string
- The result is a value of type Boolean (true or false)
- They have lower precedence than any arithmetic operators (i.e., evaluate after)

Logical Operators: andalso, orelse, not

- They can only operate on Boolean values, and result in Boolean values
- Even lower precedence than Comparison operators, and orelse is lower than and also

- or else is an inclusive or, as opposed to an exclusive or
- Truthtables of each (done in class)

If-then-else Operator

- An operator? Yes, it takes operands and evaluates to a result
- The form is if expr1 then expr2 else expr3
- expr1 must result in a Boolean value
- expr2 and expr3 must result in values of the same type
- the result of the if-then-else operator is the value of expr2 if expr1 is true, otherwise it is the value of expr3; the type of the result is the type of expr2 and expr3 some examples found here

Type consistency

- ML is a strongly typed language
- Every operator (and function) has a specific type signature, and this cannot be violated.
- For example, the operator ^ (which concatenates two strings) has a type signature of string*string->string reads as "takes a string and another string, and results in a string". order is important for the *
- What about +? We used it with both integers and reals. What is its type signature?
- it has two: int*int->int and real*real->real
- but not int*real or real*int!
- we say that + is an overloaded operator, because it has more than one signature
- Resulting types do not have to be the same as the argument types. The type signatures of the overloaded relational operator > are:
- int*int->bool
- real*real->bool
- string*string->bool
- The type signature of the if-then-else operator is bool*t*t->t where t is some type.

Type conversion (coercion)

- Values can be converted between types

- Think of the coercion as a function taking one argument of a type, and returning a value of another type.
- Easiest: integer to real: real(3) is 3.0
- What about real to integer: depends on how you want it:
- floor, ceil (ceiling), trunc (truncate)
- character to integer? integer to character?
- string to integer? real to string?