Chapter 12

Interpreters

The tools we use have a profound (and devious!) influence on our thinking habits, and, therefore, on our thinking abilities.

Edsger Dijkstra, *How do we tell truths that might hurt?*

An *interpreter* is just a program. As input, it takes a specification of a program in some language. As output, it produces the output of the input program. By designing a new interpreter, we can invent a new language.

Languages are powerful tools for thinking. Different languages encourage different ways of thinking and lead to different thoughts. Hence, inventing new languages is a powerful way for solving problems. We can solve a problem by designing a language in which it is easy to express a solution, and then implementing an interpreter for that language.

In this chapter, we explore how to implement an interpreter. We will also introduce the Python programming language, and describe an interpreter that implements a subset of the Scheme language. Implementing an interpreter further blurs the line between *data* and *programs*, that we first crossed in Chapter 3 by passing procedures as parameters and returning new procedures as results. Now that we are implementing an interpreter, all programs as just data input for our interpreter program. The meaning of the program is determined by the interpreter.

12.1 Building Languages

To implement an interpreter for a given target language we need to:

- Implement a *parser* that takes as input a string representation of a program in the target language and produces a structural parse of the input program. The parser should break the input string into its language components, and form a *parse tree* data structure that represents the input text in a structural way.

- Implement an *evaluator* that takes as input a structural parse of an input program, and evaluates that program. The evaluator should implement the target language's evaluation rules.

Section 12.2 describes our parser implementation. Section 12.3 describes the implementation of our evaluator. The next subsection introduces the target language for our interpreter. The following subsection introduces Python, the language we will use to implement our interpreter, and for most of the rest of the programs in this book.

12.1.1 Charme

Our target language is a simple subset of Scheme we will call *Charme*.[1]

The Charme language we will implement is simple, but powerful enough to express all computations (that is, it is a universal programming language). Its evaluation rules are a subset of the Scheme evaluation rules. It includes the application expression, conditional expression, lambda expression, name expression, and definitions. It supports integral numbers, and provides the basic arithmetic and comparison primitives with the same meanings as they have in Scheme.

[1]The original name of Scheme was "Schemer", a successor to the languages "Planner" and "Conniver". Because the computer on which "Schemer" was implemented only allowed six-letter file names, its name was shortened to "Scheme". In that spirit, we name our snake-charming language, "Charmer" and shorten it to Charme. Depending on the programmer's state of mind, the language name can be pronounced either "charm" or "char me".

12.1.2 Python

We could implement our Charme interpreter using Scheme, but Python[2] is a popular programming language initially designed by Guido van Rossum in 1991. Python is widely used to develop dynamic web applications and as a scripting language for applications (including the game Civilization IV). Python was used to manage special effects production for Star Wars: Episode II, and is used extensively at Google, reddit.com, and NASA.[3]

Like Scheme, Python is a *universal programming language*. We will define this more formally in Chapter **??**, but for now, think of it as meaning that both Python and Scheme and express all computations. There is no Scheme program that does not have an equivalent Python program, and every Python program has an equivalent Scheme program. One piece of evidence that every Scheme program has an equivalent Python program is the interpreter we describe in this chapter. Since we can implement a Scheme interpreter in Python, we know we can express every computation that can be expressed by a Scheme program with an equivalent Python program (that is, the Scheme interpreter implemented in Python with the original Scheme program as input).

The grammar for Python is quite different from the Scheme grammar, so Python programs look very different from Scheme programs. In most respects, however, the evaluation rules are quite similar to the evaluation rules for Scheme. This section does not describe the entire Python language, but instead introduces what we need as we need it. See the Python guide for a more comprehensive introduction to the Python programming language.

12.2 Parser

The parser takes as input a Charme program string, and produces as output a Python list that encodes the structure of the input program. Charme's syntax makes implementing the parser fairly simple, since the grammar rules for Scheme are very simple. It is not necessary to build a complex parser, since we can break an expression into its components just by using the parentheses and whitespace.

[2]The name *Python* alludes to Monty Python's Flying Circus.

[3]See http://python.org/Quotes.html for descriptions of Python uses.

The parser needs to balance the open and close parentheses that enclose expressions. The parsed program can be represented as a list.

For example, if the input string is "(define square (lambda (x) (* x x)))" the output of the parser should be:

```
['define',
 'square',
 [  'lambda',
    ['x'],
    ['*', 'x', 'x']
 ]
]
```

Python provides a list datatype similar to the Scheme list datatype. Lists are denoted using square brackets, `[` and `]`. Hence, the output parse structure is a list containing three elements, the first is the keyword `'define'`, the second is the name `'square'`, and the third is a list containing three elements, `['lambda', ['x'], ['*', 'x', 'x']]`, the third of which is itself a list containing three elements.

We divide the job of parsing into two procedures that are combined to solve the problem of transforming an input string into a list describing the input program's structure. The first part is the *scanner*. A scanner *tokenizes* an input string. Its input is the input string in the target programming language, and its output is a list of the tokens in that string.

A token is an indivisible syntactic unit. For example, in the example above the tokens are `(`, `define`, `square`, `(`, `lambda`, `(`, `x`, `)`, `(`, `*`, `x`, `x`, `)`, `)`, and `)`. The tokens are separated by whitespace (spaces, tabs, and newlines). The left and right parentheses are tokens by themselves.

12.2.1 Tokenizing

The `tokenize` procedure below takes as input a string `s` in the Charme target language, and produces as output a list of the tokens in `s`. We describe it in detail below, but attempt to understand it on your own first.

```
def tokenize(s):
    current = ''
    tokens = []
    for c in s:
        if c.isspace():
            if len(current) > 0:
                tokens.append(current)
                current = ''
        elif c in '()':
            if len(current) > 0:
                tokens.append(current)
                current = ''
            tokens.append(c)
        else:
            current = current + c

    if len(current) > 0:
        tokens.append(current)
    return tokens
```

Unlike in Scheme, the whitespace (such as new lines) has meaning in Python. Statements cannot be separated into multiple lines, and only one statement may appear on a single line. Indentation within a line also matters. Instead of using parentheses to provide code structure, Python uses the indentation to group statements into blocks. The Python interpreter will report an error if the indentation of the code does not match its structure.

12.2.2 Statements and Expressions

Whereas Scheme programs were composed of expressions and definitions, Python programs are mostly sequences of statements. Unlike expressions which (mostly) evaluate to values, a statement does not have a value. The emphasis on statements reflects (and impacts) the style of programming used with Python. It is more imperative than that used with Scheme: instead of composing expressions in ways that pass the result of one expression as an operand to the next expression, Python programs consist of a sequence of statements, each of which alters the state in some way towards reaching the goal state. Nevertheless, it is possible

(but not recommended) to program in Scheme using an imperative style (emphasizing `begin` and `set!` expressions), and it is possible (but not recommended) to program in Python using a functional style (emphasizing procedure applications and eschewing assignment statement).

Defining a procedure in Python is similar to defining a procedure in Scheme, except the grammar rule is different:

ProcedureDefinition	::⇒	**def** ***Name*** **(** *Parameters* **) :** *Block*
Parameters	::⇒	ϵ
Parameters	::⇒	*SomeParameters*
SomeParameters	::⇒	***Name***
SomeParameters	::⇒	***Name*** **,** *SomeParameters*
Block	::⇒	<**newline**> indented(*Statements*)
Statements	::⇒	*Statement* <**newline**> *MoreStatements*
MoreStatements	::⇒	*Statement* <**newline**> *MoreStatements*
MoreStatements	::⇒	ϵ

Since whitespace matters in Python, we include newlines (<**newline**>) and indentation in our grammar. We use

indented(*elements*)

to indicate that the *elements* are indented. For example, the rule for *Block* is a newline, followed by one or more statements. The statements are all indented one level inside the block's indentation. This means it is clear when the block's statements end because the next line is not indented to the same level.

The evaluation rule for a procedure definition is similar to the rule for the Scheme procedure definition. It defines the given name as a procedure that takes the parameters as it inputs and has the body given by the statement block.

So, our procedure definition

```
def tokenize(s):
    ...
```

defines a procedure named `tokenize` that takes a single parameter, `s`.

The body of the procedure uses several different types of Python statements. Following Python's more imperative style, five of the 12 statements are assignment statements:

Statement	::⇒	*AssignmentStatement*
AssignmentStatement	::⇒	*Target* **=** *Expression*
Target	::⇒	***Name***

For now, we use only a *Name* as the left side of an assignment, but since other constructs can appear on the left side of an assignment statement, we introduce the nonterminal *Target* for which additional rules can be defined to encompass other possible assignees.

The evaluation rule for an assignment statement is similar to Scheme's evaluation rule for set expressions: the meaning of `x = e` in Python is similar to the meaning of `(set! x e)` in Scheme, except that the target *Name* need not exist before the assignment.

> **Evaluation Rule: Assignment.** To evaluate an assignment statement, evaluate the expression, and assign the value of the expression to the place identified by the target. If no such place exists, create a new place with that name.

Like Scheme, Python supports many different kinds of expressions. The `tokenize` procedure uses literals, primitives, applications, arithmetic, and comparison expressions.

Since Python does not use parentheses to group expressions, the grammar provides the grouping by breaking down expression in several steps. This defines an order of *precedence* for parsing expressions. If a complex expression includes many expressions, the grammar specifies how they will be grouped. For example, consider the expression `3+4*5`. In Scheme, the expressions `(+ 3 (* 4 5))` and `(* (+ 3 4) 5)` are clearly different and the parentheses group the subexpressions. The meaning of the Python expression `3+4*5` is `(+ 3 (* 4 5))`, that is, it evaluates to 23. The expression `4*5+3` also evaluates to 23.

This makes the Python grammar rules more complex since they must deal with * and + differently, but it makes the meaning of Python expressions match our familiar mathematical interpretation, without needing all the parentheses we need in Scheme expressions. The way this is done is by defining the grammar rules so an *AddExpression* can contain a *MultExpression* as one of its subexpressions, but a *MultExpression* cannot contain an *AddExpression*. This makes the multiplication operator have *higher precedence* than the addition operator. If an expression contains both + and * operators, the * operator attaches to its operands first. The replacement rules that happen first have lower precedence, since their components must be built from the remaining pieces.

Here is a subset of the grammar rules for Python expressions for comparison, multiplication, and addition expressions that achieves this:

Expression	::⇒	*ComparisonExpr*
ComparisonExpr	::⇒	*AddExpression*
ComparisonExpr	::⇒	*ComparisonExpr Comparator ComparisonExpr*
Comparator	::⇒	< | > | **==** | **<=** | **>=**
AddExpression	::⇒	*MultExpression*
AddExpression	::⇒	*AddExpression* **+** *MultExpression*
AddExpression	::⇒	*AddExpression* **-** *MultExpression*
MultExpression	::⇒	*MultExpression* ***** *PrimaryExpression*
MultExpression	::⇒	*PrimaryExpression*
PrimaryExpression	::⇒	*Literal*
PrimaryExpression	::⇒	***Name***
PrimaryExpression	::⇒	**(** *Expression* **)**

Note that the last rule allows parentheses to be used to group expressions. For example, `(3 + 4) * 5` would be parsed as the primary expression, `(3 + 4)`, times `5`, so it evaluates to 35.

A *Literal* can be a numerical constant. Numbers in Python are similar (but not identical) to numbers in Scheme. In the example program, we use the integer literal `0`.

A *PrimaryExpression* can also be a name, similar to names in Scheme. The evaluation rule for a name in Python is similar to the stateful rule for evaluating a name

in Scheme[4].

Exercise 12.1. Do comparison expressions have higher or lower precedence than addition expressions? Explain why using the grammar rules. ♢

Exercise 12.2. What do the following Python expressions evaluate to:

a. `1 + 2 + 3 * 4`

b. `3 > 2 + 2`

c. `3 * 6 >= 15 == 12`

d. `(3 * 6 >= 15) == True`

♢

12.2.3 Lists

Python provides a list datatype similar to lists in Scheme, except instead of building list from simpler parts (that is, using `cons` pairs in Scheme), the Python list type is provided as a built-in datatype. Lists are denoted in Python using square brackets. For example, `[]` denotes an empty list, and `[1, 2]` denotes a list containing two elements. In the `tokenize` procedure, we use `tokens = []` to initialize `tokens` to an empty list.

Elements can be selected from a list using the list subscription expression:

PrimaryExpression	::⇒	*SubscriptExpression*
SubscriptExpression	::⇒	*PrimaryExpression* **[** *Expression* **]**

If the first primary expression evaluates to a list, the subscript expression selects the element indexed by value of the inner expression from the list. For example,

[4]There are some subtle differences and complexities (see Section 4.1 of the Python Reference Manual, however, which we do not go into here.

```
>>> a = [1, 2, 3]
>>> a[0]
1
>>> a[1+1]
3
>>> a[3]
IndexError: list index out of range
>>> [1,2][1]
2
```

So, the expression `p[0]` in Python is analogous to `(car p)` in Scheme.

We can also use negative selection indexes to select elements from the back of the list. The expression `p[-1]` selects the last element in the list `p`.

A subscript expression can also select a range of elements from the list:

SubscriptExpression	::⇒	*PrimaryExpression* **[** $Bound_{Low}$ **:** $Bound_{High}$ **]**
Bound	::⇒	*Expression* $\mid \epsilon$

The subscript expression evaluates to a list containing the elements between the low bound and the high bound. If the low bound is missing, the low bound is the beginning of the list. If the high bound is missing, the high bound is the end of the list. For example,

```
>>> a = [1, 2, 3]
>>> a[:1]
[1]
>>> a[1:]
[2, 3]
>>> a[4-2:3]
[3]
>>> a[:]
[1, 2, 3]
```

So, the expression `p[1:]` in Python is analogous to `(cdr p)` in Scheme.

Python lists are mutable (the value of a list can change after it is created). We can use list subscripts as the targets for an assignment expression:

Target ::⇒ *SubscriptExpression*

For example,

```
>>> a = [1, 2, 3]
>>> a[0] = 7
>>> a
[7, 2, 3]
>>> a[1:4] = [4, 5, 6]
>>> a
[7, 4, 5, 6]
>>> a[1:]  = [6]
>>> a
[7, 6]
```

Note that assignments can not only be used to change the values of elements in the list, but also to change the length of the list.

12.2.4 Strings

The other datatype used in `tokenize` is the string datatype, named `str` in Python. Strings are immutable strings of characters. Unlike lists, which are mutable, once a string is created its value cannot change. The value of a variable that is a string can change by assigning a new string object to that variable, but this does not change the value of the original string object.

Strings are enclosed in quotes, which can be either single quotes (e.g., `'hello'`) or double quotes (e.g., `"hello"`). In our example program, we use the assignment expression, `current = ''`, to initialize the value of `current` to the empty string. The input, `s`, is a string object.

12.2.5 Objects and Methods

In Python, every data value, including lists and strings, is an object. This means the way we manipulate data is to invoke methods on objects. The list datatype provides methods for manipulating and observing lists. The grammar rules for calling procedures are given below.

PrimaryExpression	::⇒	*CallExpression*
CallExpression	::⇒	*PrimaryExpression* **(** *ArgumentList* **)**
ArgumentList	::⇒	*SomeArguments*
ArgumentList	::⇒	ϵ
SomeArguments	::⇒	*Expression*
SomeArguments	::⇒	*Expression* **,** *SomeArguments*

To invoke a method we use the same rules, but the *PrimaryExpression* of the *CallExpression* specifies an object and method:

PrimaryExpression	::⇒	*AttributeReference*
AttributeReference	::⇒	*PrimaryExpression* **.** ***Name***

The name *AttributeReference* is used since the same syntax is used for accessing the internal state of objects as well.

The `tokenize` procedure includes five method applications, four of which are `tokens.append(current)`. The object reference is `tokens`, the list we use to keep track of the tokens in the input. The list datatype provides the `append` method. It takes one parameter and adds that value to the end of the list. Hence, these invocations add the value of `current` to the end of the `tokens` list.

The other method invocation is `c.isspace()` where `c` is a string consisting of one character in the input. The `isspace` method for the string datatype returns true if the input string is non-empty and all characters in the string are whitespace (either spaces, tabs, or newlines).

The `tokenize` procedure also uses the built-in function `len`. The `len` function takes as input an object of a collection datatype (including a list or a string),

and outputs the number of elements in the collection. In `tokenize`, we use `len(current)` to find the number of elements in the current token. Note that `len` is a procedure, not a method. The input object is passed in as a parameter.

12.2.6 Control Statements

Python provides control statements for making decisions and looping. Other than the first two initializations, and the final two statements, the bulk of the `tokenize` procedure is a `for` statement. The `for` statement provides a way of iterating through a set of values, carrying out a body block for each value.

Statement	::⇒	ForStatement
ForStatement	::⇒	**for** *Target* **in** *Expression* **:** *Block*

The *Target* (as was used in the assignment statement) is typically a variable name. The value of the *Expression* is a collection of elements. To evaluate a for statement, each value of the *Expression* collection is assigned to the *Target* in order, and the *Block* is evaluated once for each value. The for statement in `tokenize` header is `for c in s`. The string `s` is the input string, a collection of characters. So, the loop will repeat once for each character in `s`, and the value of `c` will be a string consisting of a single character, each character in the input string, in turn.

Python's if statement is similar to both the if and conditional expressions in Scheme:

Statement	::⇒	IfStatement
IfStatement	::⇒	**if** $Expression_{Predicate}$ **:** *Block Elifs OptElse*
Elifs	::⇒	ϵ
Elifs	::⇒	**elif** $Expression_{Predicate}$ **:** *Block Elifs*
OptElse	::⇒	ϵ
OptElse	::⇒	**else :** *Block*

The evaluation rule is similar to Scheme's conditional expression. First, the $Expression_{Predicate}$ of the **if** is evaluated. If it evaluates to a true value, the consequence *Block* is evaluated, and none of the rest of the *IfStatement* is evaluated. Otherwise, each of the

elif predicates is evaluated in order. If one evaluates to a true value, its *Block* is evaluated and none of the rest of the *IfStatement* is evaluated. If none of the *elif* predicates evaluates to a true value, the **else** *Block* is evaluated (if there is one). The main *IfStatement* in `tokenize` is:

```
if c.isspace():
    if len(current) > 0:
        tokens.append(current)
        current = ''
elif c in '()':
    if len(current) > 0:
        tokens.append(current)
        current = ''
    tokens.append(c)
else:
    current = current + c
```

The `if` predicate tests if the current character is a space. If so, the end of the current token has been reached. The consequence *Block* is itself an *IfStatement*. If the current token has at least one character, it is appended to the list of tokens in the input string and the current token is reset to the empty string. This *IfStatement* has no *elif* or *else* clauses, so if the predicate is false, there is nothing to do. If the predicate for the main *IfStatement* is false, evaluation proceeds to the *elif* clause. The predicate for this clauses tests if `c` is in the set of characters given by the literal string `'()'`. That is, it is true if `c` is either an open or close parentheses. As with spaces, a parenthesis ends the previous token, so the first statement in the *elif* clauses is identical to the first consequent clause. The difference is unlike spaces, we need to keep trace of the parentheses, so it is added to the token list by `tokens.append(c)`. The final clause is an *else* clause, so its body will be evaluated if neither the `if` or `elif` predicate is true. This means the current character is not a space or a parenthesis, so it is some other character. It should be added to the current token. This is done by the assignment expression, `current = current + c`. The addition operator in Python works on strings as well as numbers (and some other datatypes). For strings, it concatenates the operands into a new string. Recall that strings are immutable, so there is no equivalent to the `append` method. Instead, appending a character to a string involves creating a new string object.

Finally, we consider the last two statements in our `tokenize` procedure:

```
if len(current) > 0:
    tokens.append(current)
return tokens
```

These statements are not indented to the level inside the *ForStatement*. Hence, they will be evaluated after all iterations of the for loop have finished. The first statement is an if statement that adds the current token (if it is non-empty) to the list of tokens. This would happen, for example, if the last character in the input string is not a parenthesis or a space. The final statement is a *ReturnStatement*:

Statement	::⇒	ReturnStatement
ReturnStatement	::⇒	**return** *Expression*

A return statement finishes execution of a procedure body and returns the value of the *Expression* to the caller as the result. In this instance the return statement is the last statement in the procedure body, but return statements may appear anywhere inside a procedure body.

12.2.7 Parsing

The next step is to take the list of tokens and produce a data structure that encodes the structure of the input program. Since the Scheme language is built from simple s-expressions, it will be adequate to use a list data structure as the result of the parse. But, unlike the list returned by `tokenize` which is a flat list containing the tokens in order, the list returned by `parse` is a structured list that may have lists (and lists of lists, etc.) as elements. The input to `parse` is a string in the target language. The output is a list of the s-expressions in the input. Here are some examples:

```
>>> parse("150")
['150']
>>> parse("(+ 1 2)")
```

```
[['+', '1', '2']]
>>> parse("(+ 1 (* 2 3))")
[['+', '1', ['*', '2', '3']]]
>>> parse("(define square (lambda (x) (* x x)))")
[['define', 'square', ['lambda', ['x'], ['*', 'x', 'x']]]]
>>> parse("(+ 1 2) (+ 3 4)")
[['+', '1', '2'], ['+', '3', '4']]
```

Note that the parentheses are no longer included as tokens in the result, but their presence in the input string determines the structure of the result.

Here is the definition of `parse`:

```
def parse(s):
    def parsetokens(tokens, inner):
       res = []
       while len(tokens) > 0:
          current = tokens.pop(0)
          if current == '(':
             res.append (parsetokens(tokens, True))
          elif current == ')':
             if inner:
                return res
             else:
                error("Unmatched close paren: " + s)
                return None
          else:
             res.append(current)

       if inner:
          error ("Unmatched open paren: " + s)
          return None
       else:
          return res

    return parsetokens(tokenize(s), False)
```

It implements what is known as a *recursive descent* parser. The main `parse` procedure defines the `parsetokens` helper procedure and returns the result of calling it with inputs that are the result of tokenizing the input string and the Boolean literal `False`: `return parsetokens(tokenize(s), False)`.

The `parsetokens` procedure takes two inputs: `tokens`, a list of tokens (that results from the `tokenize` procedure); and `inner`, a Boolean that indicates whether the parser is inside a parenthesized expression. The value of `inner` is `False` for the initial call since the parser starts outside a parenthesized expression. All of the recursive calls result from encountering a `'('`, so the value passed as `inner` is `True` for all the recursive calls.

The body of the `parsetokens` procedure initializes `res` to an empty list that will be used to store the result. Then, the `while` statement iterates as long as the token list contains at least one element. The first statement of the `while` statement block assigns `tokens.pop(0)` to `current`. The `pop` method of the list takes a parameter that selects an element from the list. The selected element is returned as the result. The `pop` method also mutates the list object by removing the selected element. So, `tokens.pop(0)` returns the first element of the `tokens` list and removes that element from the list. This is similar to `(cdr tokens)` with one big difference: the `tokens` object is modified by the call. This is essential to the parser making progress: every time the `tokens.pop(0)` expression is evaluated the number of elements in the token list is reduced by one.

If the `current` token is an open parenthesis, `parsetokens` is called recursively to parse the inner s-expression (that is, all the tokens until the matching close parenthesis). The result is a list of tokens, which is appended to the result. If the `current` token is a close parenthesis, the behavior depends on whether or not the parser is parsing an inner s-expression. If it is inside an s-expression (that is, an open parenthesis has been encountered with no matching close parenthesis yet), the close parenthesis closes the inner s-expression, and the result is returned. If it is not in an inner expression, the close parenthesis has no matching open parenthesis so a parse error is reported. The `else` clause deals with all non-parentheses tokens by appending them to the result list.

12.3 Evaluator

The evaluator takes a list representing a parsed program fragment in Charme and an environment, and outputs the result of evaluating the input code in the input environment. The evaluator implements the evaluation rules for the target language.

The core of the evaluator is the procedure `meval`:

```
def meval(expr, env):
    if isPrimitive(expr):
       return evalPrimitive(expr)
    elif isConditional(expr):
       return evalConditional(expr, env)
    elif isDefinition(expr):
       evalDefinition(expr, env)
    elif isName(expr):
       return evalName(expr, env)
    elif isLambda(expr):
       return evalLambda(expr, env)
    elif isApplication(expr):
       return evalApplication(expr, env)
    else:
       error ("Unknown expression type: " + str(expr))
```

The if statement matches the input expression with one of the expression types (or the definition) in the Charme language, and returns the result of applying the corresponding evaluation procedure (if the input is a definition, no value is returned since definitions do not produce an output value). We next consider each evaluation rule in turn.

12.3.1 Primitives

Charme supports three kinds of primitives: natural numbers, Boolean constants, and primitive procedures.

If the expression is a number, it is a string of digits. The `isNumber` procedure evaluates to `True` if and only if its input is a number:

```
def isNumber(expr):
    return isinstance(expr, str) and expr.isdigit()
```

Here, we use the built-in function `isinstance` to check if `expr` is of type `str`. The `and` expression in Python evaluates similarly to the Scheme `and` special form: the left operand is evaluated first; if it evaluates to a false value, the value of the and expression is that false value. If it evaluates to a true value, the right operand is evaluated, and the value of the and expression is the value of its right operand. This evaluation rule means it is safe to use `expr.isdigit()` in the right operand, since it is only evaluated if the left operand evaluated to a true value, which means `expr` is a string.

To evaluate a number primitive, we need to covert the string representation to a number of type `int`. The `int(s)` constructor takes a string as its input and outputs the corresponding integer:

```
def evalPrimitive(expr):
    if isNumber(expr):
        return int(expr)
    else:
        return expr
```

The `else` clause means that all other primitives (in Charme, this is only primitive procedures and Boolean constants) self-evaluate: the value of evaluating a primitive is itself.

Primitive procedures are defined using Python procedures. Hence, the `isPrimitiveProcedure` procedure is defined using `callable`, a procedure that returns true only for objects that are callable (such as procedures and methods):

```
def isPrimitiveProcedure(expr):
    return callable(expr)
```

Here is the `primitivePlus` procedure that is associated with the + primitive procedure:

```
def primitivePlus (operands):
    if (len(operands) == 0):
       return 0
    else:
       return operands[0] + primitivePlus (operands[1:])
```

The input is a list of operands. Since a procedure is applied only after all subexpressions are evaluated (according to the Scheme evaluation rule for an application expression), there is no need to evaluate the operands: they are already the evaluated values. For numbers, the values are Python integers, so we can use the Python + operator to add them. To provide the same behavior as the Scheme primitive + procedure, we define our Charme primitive + procedure to evaluate to 0 when there are no operands, and otherwise, recursively add all of the operand values.

The other primitive procedures are defined similarly.

```
def primitiveTimes (operands):
    if (len(operands) == 0):
       return 1
    else:
       return operands[0] * primitiveTimes (operands[1:])

def primitiveMinus (operands):
    if (len(operands) == 1):
       return -1 * operands[0]
    elif len(operands) == 2:
       return operands[0] - operands[1]
    else:
       evalError("- expects 1 or 2 operands, given %s: %s"
                 % (len(operands), str(operands)))
```

```
def primitiveEquals (operands):
    checkOperands (operands, 2, "=")
    return operands[0] == operands[1]

def primitiveZero (operands):
    checkOperands (operands, 1, "zero?")
    return operands[0] == 0

def primitiveGreater (operands):
    checkOperands (operands, 2, ">")
    return operands[0] > operands[1]

def primitiveLessThan (operands):
    checkOperands (operands, 2, "<")
    return operands[0] < operands[1]
```

The `checkOperands` procedure reports an error if a primitive procedure is applied to the wrong number of operands:

```
def checkOperands(operands, num, prim):
    if (len(operands) != num):
       evalError("Primitive %s expected %s operands, given %s: %s"
                 % (prim, num, len(operands), str(operands)))
```

12.3.2 Conditionals

Charme provides a conditional expression with an evaluation rule identical to the Scheme conditional expression. We recognize a conditional expression by the `cond` token at the beginning of the expression:

```
def isSpecialForm(expr, keyword):
```

```
    return isinstance(expr, list) \
           and len(expr) > 0 and expr[0] == keyword

def isConditional(expr):
    return isSpecialForm(expr, 'cond')
```

We use the
continuation character in the return statement of `isSpecialForm` to split one expression into multiple lines. Recall that newlines matter in Python, so it would be an error to split a statement onto multiple likes without using the continuation character.

To evaluate a conditional, we need to follow the evaluation rule:

```
def evalConditional(expr, env):
    assert isConditional(expr)
    if len(expr) <= 2:
        evalError ("Bad conditional expression: %s" % str(expr))
    for clause in expr[1:]:
        if len(clause) != 2:
            evalError ("Bad conditional clause: %s" % str(clause))
        predicate = clause[0]
        result = meval(predicate, env)
        if result:
            return meval(clause[1], env)
    evalError ("No conditional predicate is true: %s"
               % (expr))
    return None
```

12.3.3 Definitions and Names

To evaluate definitions and names we need to represent environments. A definition adds a name to a frame, and a name expression evaluates to the value associated with a name.

We will use a Python class to represent an environment. As in Chapter 10, a class packages state and procedures that manipulate that state. In Scheme, we

needed to use a message-accepting procedure to do this. Python provides the class construct to support it directly. We define the `Environment` class for representing an environment. It has internal state for representing the parent (itself an `Environment` or `None`, Python's equivalent to `null` for the global environment's parent), and for the frame. The Python dictionary datatype provides a convenient way to implement a frame. It is a lookup-table where values are associated with keys. Curly brackets are used to denote a dictionary. The `__init__` procedure constructs a new object. It initializes the frame of the new environment to the empty dictionary using `self._frame = { }`. The `addVariable` method either defines a new variable or updates the value associated with a variable. Using the dictionary datatype, we can do this with a simple assignment statement. The `lookupVariable` method first checks if the frame associated with this environment has a key associated with the input `name`. If it does, the value associated with that key is the value of the variable and that value is returned. Otherwise, if the environment has a parent, the value associated with the name is the value of looking up the variable in the parent environment. This directly follows from the stateful Scheme evaluation rule for name expressions. The `else` clause addresses the situation where the name is not found and there is no parent environment (since we have already reached the global environment) by reporting an evaluation error indicating an undefined name.

```
class Environment:
    def __init__(self, parent):
        self._parent = parent
        self._frame = { }
    def addVariable(self, name, value):
        self._frame[name] = value
    def lookupVariable(self, name):
        if self._frame.has_key(name):
            return self._frame[name]
        elif (self._parent):
            return self._parent.lookupVariable(name)
        else:
            evalError("Undefined name: %s" % (name))
```

Once the `Environment` class is defined, implementing the evaluation rules for definitions and name expressions is straightforward.

```
def isDefinition(expr):
    return isSpecialForm(expr, 'define')

def evalDefinition(expr, env):
    assert isDefinition(expr)
    if len(expr) != 3:
        evalError ("Bad definition: %s" % str(expr))
    name = expr[1]
    if isinstance(name, str):
        value = meval(expr[2], env)
        env.addVariable(name, value)
    else:
        evalError ("Bad definition: %s" % str(expr))

def isName(expr):
    return isinstance(expr, str)

def evalName(expr, env):
    assert isName(expr)
    return env.lookupVariable(expr)
```

12.3.4 Procedures

The result of evaluating a lambda expression is a procedure. Hence, to define the evaluation rule for lambda expressions we need to define a class for representing user-defined procedures. It needs to record the parameters, procedure body, and defining environment:

```
class Procedure:
```

```
    def __init__(self, params, body, env):
        self._params = params
        self._body = body
        self._env = env
    def getParams(self):
        return self._params
    def getBody(self):
        return self._body
    def getEnvironment(self):
        return self._env
    def toString(self):
        return "<Procedure %s / %s>" \
               % (str(self._params), str(self._body))
```

Using this, we can define the evaluation rule for lambda expressions to create a `Procedure` object:

```
def isLambda(expr):
    return isSpecialForm(expr, 'lambda')

def evalLambda(expr,env):
    assert isLambda(expr)
    if len(expr) != 3:
        evalError ("Bad lambda expression: %s" % str(expr))
    return Procedure(expr[1], expr[2], env)
```

12.3.5 Application

The evaluators circularity comes from the way evaluation and application are defined recursively. To perform an application, we need to evaluate all the subexpressions of the application expression, and then apply the result of evaluating the first subexpression to the values of the other subexpressions.

```
def isApplication(expr):
  # requires all special forms checked first
    return isinstance(expr, list)

def evalApplication(expr, env):
    subexprs = expr
    subexprvals = map (lambda sexpr: meval(sexpr, env), \
                       subexprs)
    return mapply(subexprvals[0], subexprvals[1:])
```

The `evalApplication` procedure uses the built-in `map` procedure, which is similar to Scheme's `map` procedure. The first parameter to `map` is a procedure constructed using a lambda expression (identical in meaning, but not in syntax, to Scheme's lambda expression); the second parameter is the list of subexpressions.

The `mapply` procedure implements the application rules. If the procedure is a primitive, it "just does it": it applies the primitive procedure to its operands. To apply a constructed procedure (represented by an object of the `Procedure` class) it follows the stateful application rule for applying constructed procedures: it creates a new environment, puts variables in that environment for each parameter and binds them to the corresponding operand values, and evaluates the procedure body in the new environment.

```
def mapply(proc, operands):
    if (isPrimitiveProcedure(proc)):
        return proc(operands)
    elif isinstance(proc, Procedure):
     params = proc.getParams()
     newenv = Environment(proc.getEnvironment())
     if len(params) != len(operands):
        evalError ("Parameter length mismatch: " \
                   "%s given operands %s" %
                    (proc.toString(), str(operands)))
     for i in range(0, len(params)):
         newenv.addVariable(params[i], operands[i])
     return meval(proc.getBody(), newenv)
```

```
    else:
        evalError("Application of non-procedure: %s" % (proc))
```

12.3.6 Finishing the Interpreter

To finish the interpreter, we define the `evalLoop` procedure that sets up the global environment and provides a simple user interface to the interpreter. To initializes the global environment, we need to create an environment with no parent and place variables in it corresponding to the primitives in Charme:

```
def initializeGlobalEnvironment():
    global globalEnvironment
    globalEnvironment = Environment(None)
    globalEnvironment.addVariable('#t', True)
    globalEnvironment.addVariable('#f', False)
    globalEnvironment.addVariable('+', primitivePlus)
    globalEnvironment.addVariable('-', primitiveMinus)
    globalEnvironment.addVariable('*', primitiveTimes)
    globalEnvironment.addVariable('=', primitiveEquals)
    globalEnvironment.addVariable('zero?', primitiveZero)
    globalEnvironment.addVariable('>', primitiveGreater)
    globalEnvironment.addVariable('<', primitiveLessThan)
```

The evaluation loop reads a string from the user using the Python built-in procedure `raw_input`. It uses `parse` to covert that string into a structured list representation. Then, evaluates each expression in the input string using `meval`.

```
def evalLoop():
    initializeGlobalEnvironment()
    while True:
```

```
expr = raw_input("Charme> ")
if expr == 'quit':
    break
exprs = sexpr.parse(expr)
for expr in exprs:
    print meval(expr, globalEnvironment)
```

12.4 Summary

Languages are tools for thinking, as well as means to express executable programs. A programming language is defined by its grammar and evaluation rules. To implement a language, we need to implement a parser that carries out the grammar rules and an evaluator that implements the evaluation rules. Once we have an interpreter, we can change the meaning of our language by changing the evaluation rules. In the next chapter, we will see some examples that illustrate the value of being able to extend and change a language.