3.4 Interpreters for Languages with Combination

We now embark on a tour of the technology by which languages are established in terms of other languages. *Metalinguistic abstraction* — establishing new languages — plays an important role in all branches of engineering design. It is particularly important to computer programming, because in programming not only can we formulate new languages but we can also implement these languages by constructing interpreters. An interpreter for a programming language is a function that, when applied to an expression of the language, performs the actions required to evaluate that expression.

We will first define an interpreter for a language that is a limited subset of Scheme, called Calculator. Then, we will develop a sketch of an interpreter for Scheme as a whole. The interpreter we create will be complete in the sense that it will allow us to write fully general programs in Scheme. To do so, it will implement the same environment model of evaluation that we introduced for Python programs in Chapter 1.

Many of the examples in this section are contained in the companion Scheme-Syntax Calculator example, as they are too complex to fit naturally in the format of this text.

3.4.1 A Scheme-Syntax Calculator

The Scheme-Syntax Calculator (or simply Calculator) is an expression language for the arithmetic operations of addition, subtraction, multiplication, and division. Calculator shares Scheme's call expression syntax and operator behavior. Addition (`+`) and multiplication (`*`) operations each take an arbitrary number of arguments:

```
> (+ 1 2 3 4)
10
> (+)
0
> (* 1 2 3 4)
24
> (*)
1
```

Subtraction (`-`) has two behaviors. With one argument, it negates the argument. With at least two arguments, it subtracts all but the first from the first. Division (`/`) has a similar pair of two behaviors: compute the multiplicative inverse of a single argument or divide all but the first into the first:

```
> (- 10 1 2 3)
4
> (- 3)
-3
> (/ 15 12)
1.25
> (/ 30 5 2)
3
> (/ 10)
0.1
```

A call expression is evaluated by evaluating its operand sub-expressions, then applying the operator to the resulting arguments:

```
> (- 100 (* 7 (+ 8 (/ -12 -3))))
16.0
```

We will implement an interpreter for the Calculator language in Python. That is, we will write a Python program that takes string lines as input and returns the result of evaluating those lines as a Calculator expression. Our interpreter will raise an appropriate exception if the calculator expression is not well formed.

3.4.2 Expression Trees

Until this point in the course, expression trees have been conceptual entities to which we have referred in describing the process of evaluation; we have never before explicitly represented expression trees as data in our programs. In order to write an interpreter, we must operate on expressions as data.

A primitive expression is just a number or a string in Calculator: either an `int` or `float` or an operator symbol. All combined expressions are call expressions. A call expression is a Scheme list with a first element (the operator) followed by zero or more operand expressions.

Scheme Pairs. In Scheme, lists are nested pairs, but not all pairs are lists. To represent Scheme pairs and lists in Python, we will define a class `Pair` that is similar to the `Rlist` class earlier in the chapter. The implementation appears in scheme_reader.

The empty list is represented by an object called `nil`, which is an instance of the class `nil`. We assume that only one `nil` instance will ever be created.

The `Pair` class and `nil` object are Scheme values represented in Python. They have `repr` strings that are Python expressions and `str` strings that are Scheme expressions.

```
>>> s = Pair(1, Pair(2, nil))
>>> s
Pair(1, Pair(2, nil))
>>> print(s)
(1 2)
```

They implement the basic Python sequence interface of length and element selection, as well as a `map` method that returns a Scheme list.

```
>>> len(s)
2
>>> s[1]
2
>>> print(s.map(lambda x: x+4))
(5 6)
```

Nested Lists. Nested pairs can represent lists, but the elements of a list can also be lists themselves. Pairs are therefore sufficient to represent Scheme expressions, which are in fact nested lists.

```
>>> expr = Pair('+', Pair(Pair('*', Pair(3, Pair(4, nil))), Pair(5, nil)))
>>> print(expr)
(+ (* 3 4) 5)
>>> print(expr.second.first)
(* 3 4)
>>> expr.second.first.second.first
3
```

This example demonstrates that all Calculator expressions are nested Scheme lists. Our Calculator interpreter will read in nested Scheme lists, convert them into expression trees represented as nested `Pair` instances (*Parsing expressions* below), and then evaluate the expression trees to produce values (*Calculator evaluation* below).

3.4.3 Parsing Expressions

Parsing is the process of generating expression trees from raw text input. A parser is a composition of two components: a lexical analyzer and a syntactic analyzer. First, the *lexical analyzer* partitions the input string into *tokens*, which are the minimal syntactic units of the language such as names and symbols. Second, the *syntactic analyzer* constructs an expression tree from this sequence of tokens. The sequence of tokens produced by the lexical analyzer is consumed by the syntactic analyzer.

Lexical analysis. The component that interprets a string as a token sequence is called a *tokenizer* or *lexical analyzer*. In our implementation, the tokenizer is a function called `tokenize_line` in scheme_tokens. Scheme tokens are delimited by white space, parentheses, dots, or single quotation marks. Delimiters are tokens, as are symbols and numerals. The tokenizer analyzes a line character by character, validating the format of symbols and numerals.

Tokenizing a well-formed Calculator expression separates all symbols and delimiters, but identifies multi-character numbers (e.g., 2.3) and converts them into numeric types.

```
>>> tokenize_line('(+ 1 (* 2.3 45))')
['(', '+', 1, '(', '*', 2.3, 45, ')', ')']
```

Lexical analysis is an iterative process, and it can be applied to each line of an input program in isolation.

Syntactic analysis. The component that interprets a token sequence as an expression tree is called a *syntactic analyzer*. Syntactic analysis is a tree-recursive process, and it must consider an entire expression that may span multiple lines.

Syntactic analysis is implemented by the `scheme_read` function in scheme_reader. It is tree-recursive because analyzing a sequence of tokens often involves analyzing a subsequence of those tokens into a subexpression, which itself serves as a branch (e.g., operand) of a larger expression tree. Recursion generates the hierarchical structures consumed by the evaluator.

The `scheme_read` function expects its input `src` to be a `Buffer` instance that gives access to a sequence of tokens. A `Buffer`, defined in the buffer module, collects tokens that span multiple lines into a single object that can be analyzed syntactically.

```
>>> lines = ['(+ 1', '   (* 2.3 45))']
>>> expression = scheme_read(Buffer(tokenize_lines(lines)))
>>> expression
```

```
Pair('+', Pair(1, Pair(Pair('*', Pair(2.3, Pair(45, nil))), nil)))
>>> print(expression)
(+ 1 (* 2.3 45))
```

The `scheme_read` function first checks for various base cases, including empty input (which raises an end-of-file exception, called `EOFError` in Python) and primitive expressions. A recursive call to `read_tail` is invoked whenever a `(` token indicates the beginning of a list.

The `read_tail` function continues to read from the same input `src`, but expects to be called after a list has begun. Its base cases are an empty input (an error) or a closing parenthesis that terminates the list. Its recursive call reads the first element of the list with `scheme_read`, reads the rest of the list with `read_tail`, and then returns a list represented as a `Pair`.

This implementation of `scheme_read` can read well-formed Scheme lists, which are all we need for the Calculator language. Parsing dotted lists and quoted forms is left as an exercise.

Informative syntax errors improve the usability of an interpreter substantially. The `SyntaxError` exceptions that are raised include a description of the problem encountered.

3.4.4 Calculator Evaluation

The scalc module implements an evaluator for the Calculator language. The `calc_eval` function takes an expression as an argument and returns its value. Definitions of the helper functions `simplify`, `reduce`, and `as_scheme_list` appear in the model and are used below.

For Calculator, the only two legal syntactic forms of expressions are numbers and call expressions, which are `Pair` instances representing well-formed Scheme lists. Numbers are *self-evaluating*; they can be returned directly from `calc_eval`. Call expressions require function application.

```
>>> def calc_eval(exp):
        """Evaluate a Calculator expression."""
        if type(exp) in (int, float):
            return simplify(exp)
        elif isinstance(exp, Pair):
            arguments = exp.second.map(calc_eval)
            return simplify(calc_apply(exp.first, arguments))
        else:
            raise TypeError(exp + ' is not a number or call expression')
```

Call expressions are evaluated by first recursively mapping the `calc_eval` function to the list of operands, which computes a list of `arguments`. Then, the operator is applied to those arguments in a second function, `calc_apply`.

The Calculator language is simple enough that we can easily express the logic of applying each operator in the body of a single function. In `calc_apply`, each conditional clause corresponds to applying one operator.

```
>>> def calc_apply(operator, args):
        """Apply the named operator to a list of args."""
        if not isinstance(operator, str):
            raise TypeError(str(operator) + ' is not a symbol')
        if operator == '+':
            return reduce(add, args, 0)
        elif operator == '-':
            if len(args) == 0:
                raise TypeError(operator + ' requires at least 1 argument')
            elif len(args) == 1:
                return -args.first
            else:
                return reduce(sub, args.second, args.first)
        elif operator == '*':
            return reduce(mul, args, 1)
        elif operator == '/':
            if len(args) == 0:
                raise TypeError(operator + ' requires at least 1 argument')
            elif len(args) == 1:
                return 1/args.first
            else:
                return reduce(truediv, args.second, args.first)
        else:
            raise TypeError(operator + ' is an unknown operator')
```

Above, each suite computes the result of a different operator or raises an appropriate `TypeError` when the wrong number of arguments is given. The `calc_apply` function can be applied directly, but it must be passed a list of *values* as arguments rather than a list of operand expressions.

```
>>> calc_apply('+', as_scheme_list(1, 2, 3))
6
>>> calc_apply('-', as_scheme_list(10, 1, 2, 3))
4
```

```
>>> calc_apply('*', nil)
1
>>> calc_apply('*', as_scheme_list(1, 2, 3, 4, 5))
120
>>> calc_apply('/', as_scheme_list(40, 5))
8.0
```

The role of `calc_eval` is to make proper calls to `calc_apply` by first computing the value of operand sub-expressions before passing them as arguments to `calc_apply`. Thus, `calc_eval` can accept a nested expression.

```
>>> print(exp)
(+ (* 3 4) 5)
>>> calc_eval(exp)
17
```

The structure of `calc_eval` is an example of dispatching on type: the form of the expression. The first form of expression is a number, which requires no additional evaluation step. In general, primitive expressions that do not require an additional evaluation step are called *self-evaluating*. The only self-evaluating expressions in our Calculator language are numbers, but a general programming language might also include strings, boolean values, etc.

Read-eval-print loops. A typical approach to interacting with an interpreter is through a read-eval-print loop, or REPL, which is a mode of interaction that reads an expression, evaluates it, and prints the result for the user. The Python interactive session is an example of such a loop.

An implementation of a REPL can be largely independent of the interpreter it uses. The function `read_eval_print_loop` below buffers input from the user, constructs an expression using the language-specific `scheme_read` function, then prints the result of applying `calc_eval` to that expression.

```
>>> def read_eval_print_loop():
        """Run a read-eval-print loop for calculator."""
        while True:
            src = buffer_input()
            while src.more_on_line:
                expression = scheme_read(src)
                print(calc_eval(expression))
```

This version of `read_eval_print_loop` contains all of the essential components of an interactive interface. An example session would look like:

```
> (* 1 2 3)
6
> (+)
0
> (+ 2 (/ 4 8))
2.5
> (+ 2 2) (* 3 3)
4
9
> (+ 1
     (- 2 3)
     (* 4 2.5))
-12
```

This loop implementation has no mechanism for termination or error handling. We can improve the interface by reporting errors to the user. We can also allow the user to exit the loop by signalling a keyboard interrupt (`Control-C` on UNIX) or end-of-file exception (`Control-D` on UNIX). To enable these improvements, we place the original suite of the `while` statement within a `try` statement. The first `except` clause handles `SyntaxError` and `ValueError` exceptions raised by `scheme_read` as well as `TypeError` and `ZeroDivisionError` exceptions raised by `calc_eval`.

```
>>> def read_eval_print_loop():
        """Run a read-eval-print loop for calculator."""
        while True:
            try:
                src = buffer_input()
                while src.more_on_line:
                    expression = scheme_read(src)
                    print(calc_eval(expression))
            except (SyntaxError, TypeError, ValueError, ZeroDivisionError) as err:
                print(type(err).__name__ + ':', err)
            except (KeyboardInterrupt, EOFError):  # <Control>-D, etc.
                print('Calculation completed.')
                return
```

This loop implementation reports errors without exiting the loop. Rather than exiting the program on an error, restarting the loop after an error message lets users revise their expressions. Upon importing the `readline` module, users can even recall their previous inputs using the up arrow or `Control-P`. The final result provides an informative error reporting interface:

```
> )
SyntaxError: unexpected token: )
> 2.3.4
ValueError: invalid numeral: 2.3.4
> +
TypeError: + is not a number or call expression
> (/ 5)
TypeError: / requires exactly 2 arguments
> (/ 1 0)
ZeroDivisionError: division by zero
```

As we generalize our interpreter to new languages other than Calculator, we will see that the `read_eval_print_loop` is parameterized by a parsing function, an evaluation function, and the exception types handled by the `try` statement. Beyond these changes, all REPLs can be implemented using the same structure.

Continue: 3.5 Interpreters for Languages with Abstraction