3.4 组合语言的解释器

::: details INFO
译者： silver

来源：3.4 Interpreters for Languages with Combination

对应：HW 06
:::

我们现在开始一场技术之旅，通过这种技术，编程语言可以建立在其它语言之上。元语言抽象——建立新的语言——其在工程设计的各个分支中都发挥着重要作用。这对计算机编程尤为重要，因为在编程中，我们不仅可以制定新的语言，还可以通过构造解释器来实现这些语言。编程语言的解释器是一种函数，当应用于该语言的表达式时，它执行计算该表达式所需的操作。

我们将首先定义一种语言的解释器，它是 Scheme 的一个有限子集，称为计算器语言。然后，我们将为 Scheme 整体开发一个简略的解释器。我们创建的解释器将是完整的，因为它允许我们用 Scheme 编写完全通用的程序。为此，它将实现我们在第 1 章为 Python 程序引入的相同的运算模型。

由于本节中的许多示例过于复杂，无法自然融入本文的格式中。因此，它们都将包含在配套的 Scheme-Syntax Calculator 网站中。

3.4.1 基于 Scheme 语法的计算器

基于 Scheme 语法的计算器（或简称计算器语言）是一种用于加法、减法、乘法和除法运算的表达式语言。计算器语言使用 Scheme 的调用表达式语法和运算符行为。加法（`$+$`）和乘法（`$*$`）运算都可以传入任意数量的参数：

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

减法（`$-$`）具有两种行为。只传入一个参数时，它会对该值取相反数。传入至少两个参数时，它会用第一个参数减去之后的所有参数。除法（`$/$`）也有类似的两种行为：计算单个参数的倒数，或用第一个参数除以之后的所有参数：

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

一个调用表达式的求值过程是先对所有子表达式求值，然后将运算符应用于所得结果：

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

我们将在 Python 中实现计算器语言的解释器。也就是说，我们将编写一个 Python 程序，输入字符串，并返回该字符串作为计算器语言表达式的求值结果。如果计算器语言表达式不完整，我们的解释器将抛出相应的异常。

3.4.2 表达式树

到目前为止，我们在描述评估过程时引入的表达式树，还只是一个概念。我们从未显式将表达式树表示程序中的数据。为了编写解释器，我们必须将表达式作为数据进行操作。

计算器语言中的基元表达式只是数字或字符串：可能是 int，float 或运算符。所有组合表达式都是调用表达式。调用表达式是一个 Scheme 列表，第一个元素（运算符）后面有零个或多个运算表达式。

Scheme 对：在 Scheme 中，列表一定是嵌套对，但并非所有对都是列表。为了在 Python 中表示 Scheme 对和列表，我们将定义一个类似于本章前面的 `Rlist` 类的 `Pair` 类。实现代码可以在 scheme_reader 查看。

空列表由一个名为 `nil` 的对象表示，它是类 `nil` 的一个实例。我们假设运行时只有一个 `nil` 实例被创建。

`Pair` 类和 `nil` 对象是用 Python 表示的 Scheme 值。它们有代表 Python 表达式的 `repr` 字符串和代表 Scheme 表达式的 `str` 字符串。

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

它们实现了长度和元素选择的基本 Python 接口，以及返回 Scheme 列表的 `map` 方法。

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

嵌套列表。嵌套对可以表示列表，并且列表的元素本身也可以是列表。因此，嵌套对足以代表 Scheme 表达式，而后者实际上就是嵌套列表。

```
>>> 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
```

本例说明所有计算器语言表达式都是嵌套的 Scheme 列表。我们的计算器语言解释器将读入嵌套的 Scheme 列表，将其转换为表示为 `Pair` 实例的表达式树（参见下面的解析表达式章节），然后对表达式树进行求值（参见下面的计算器语言求值章节）。

3.4.3 解析表达式

解析是根据原始文本输入生成表达式树的过程。解析器由两个组件组成：词法分析器（lexical analyzer）和语法分析器（syntactic analyzer）。首先，词法分析器将输入字符串划分为 标记（token）。标记表示语言的最小语法单元，比如名称和符号。然后，语法分析器根据这个标记序列构建一个表达式树。词法分析器生成的标记序列就被语法分析器所消耗。

词法分析。将字符串解释为标记序列的组件称为分词器（tokenizer）或词法分析器。在我们的实现中，分词器是 scheme_tokens 中的函数 `tokenize_line`。Scheme 标记被空格、括号、点或单引号所分隔。分隔符同运算符和数字一样，也是标记。分词器会逐个字符地解析一行，并检验运算符和数字的格式。

对格式良好的计算器语言表达式进行分词，不仅可以分离所有运算符和分隔符，还可以识别多字符数字（例如 23），并将其转换为数字类型。

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

词法分析是一个迭代过程，可以单独作用于输入程序的每一行。

语法分析。将标记序列解析为表达式树的组件称为语法分析器。语法分析是一个树递归过程，它必须考虑可能跨越多行的整个表达式。

语法分析由 scheme_reader 中的 `scheme_read` 函数实现。它是树递归的，因为分析一个标记序列往往涉及分析这些标记的子表达式，而子表达式本身又可能是更大的表达式树的一个分支（比如操作数）。递归产生了运算器所使用的层次结构。

`scheme_read` 函数希望它的输入 `src` 是一个 `Buffer` 实例，其可以访问一系列标记。`Buffer` 在 buffer 模块中定义，它将跨多行的标记收集到一个可以进行语法分析的对象中。

```
>>> 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))
```

`scheme_read` 函数首先检查各种基本情况，包括空输入（这会引发文件结束异常，在 Python 中称为 `EOFError`）和基元表达式。每当 `(` 标记在列表的开头时，就会调用 `read_tail` 来递归解析。

`scheme_read` 的实现可以读取格式良好的 `Scheme` 列表，而这正是计算器语言所需要的。关于点列表和引号形式的解析留作练习。

信息丰富的语法错误可以极大地提高解释器的可用性。由此引发的 `SyntaxError` 异常包含对所遇问题的描述。

3.4.4 计算器语言求值

scalc 模块为计算器语言实现了一个求值器。`calc_eval` 函数将表达式作为参数并返回其值。辅助函数 `simplify`、`reduce` 和 `as_scheme_list` 的定义同样出现在模型中，并将在下文中使用。对于计算器语言，表达式仅有的两种合法语法形式是数字和调用表达式，它们都是 `Pair` 实例，并表示格式良好的 `Scheme` 列表。数字是自求值的，可以直接从 `calc_eval` 返回。调用表达式则需要调用函数。

```
>>> 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')
```

调用表达式的求值方法是，首先讲 `calc_eval` 递归地作用在操作数列表上，从而计算出参数列表。然后，调用另一个函数 `calc_apply` 来讲运算符作用于这些参数。

```
>>> 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')
```

上面，每个组件都计算不同运算符的结果，或者在传入错误数量的参数时引发适当的 `TypeError`。`calc_apply` 函数可以直接调用，但必须传入值列表作为参数而不是操作数表达式列表。

```
>>> 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
```

`calc_eval` 的作用是通过首先计算操作数子表达式的值，然后将它们作为参数传递给 `calc_apply`，从而对 `calc_apply` 进行正确的调用。因此，`calc_eval` 可以接受嵌套表达式。

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

`calc_eval` 的结构是对类型（表达式的形式）进行调度的一个示例。表达式的第一种形式是数字，它不需要额外的求值步骤。通常，不需要额外求值步骤的基元表达式称为自求值。在我们的计算器语言中，唯一的自求值表达式是数字，但通用编程语言也可能包括字符串、布尔值等。

读取 - 求值 - 打印循环：与解释器交互的一种典型方式是读取 - 求值 - 打印循环（Read-eval-print loops），或称 REPL。这是一种读取表达式、求值并为用户打印结果的交互模式。Python 交互会话就是这种循环的一个例子。

REPL 的实现可以在很大程度上独立于它所使用的解释器。下面的函数 `read_eval_print_loop` 缓冲用户输入，使用语言特定的 `scheme_read` 函数构造表达式，然后打印对该表达式应用 `calc_eval` 的结果。

```
>>> 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))
```

该版本的 `read_eval_print_loop` 包含交互式界面的所有基本组件。会话示例如下：

```
> (* 1 2 3)
6
> (+)
0
> (+ 2 (/ 4 8))
2.5
> (+ 2 2) (* 3 3)
4
9
> (+ 1
     (- 23)
     (* 4 2.5))
-12
```

该循环实现没有终止或错误处理机制。我们可以通过向用户报告错误来改进界面。我们还可以让用户通过键盘中断信号（UNIX 上为 Control-C）或文件结束异常（UNIX 上为 Control-D）来退出循环。为了实现这些改进，我们将原来的 `while` 语句放在 `try` 语句中。第一个 `except` 子句处理 `scheme_read` 引发的语法错误（`SyntaxError`）和值错误（`ValueError`）异常，以及 `calc_eval` 引发的类型错误（`TypeError`）和除零错误（`ZeroDivisionError`）异常。

```
>>> 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
```

这种循环执行方式会在不退出循环的情况下报告错误。用户可以在收到错误信息后重新启动循环，而不是在出错时退出程序，从而修改自己的表达式。在导入 `readline` 模块后，用户甚至可以使用向上箭头或 Control-P 回顾之前的输入。最终的结果是提供了一个信息丰富的错误报告界面：

```
> )
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
```

当我们将解释器推广到计算器语言以外的新语言时，我们将看到 `read_eval_print_loop` 是由解析函数、求值函数和 try 语句处理的异常类型参数化的。除了这些变化，所有 REPL 都可以使用相同的结构来实现。