《函数式语言程序设计》课程作业文档

蒋梦青 2014013443
钟皓曦 2014011384
叶　佩 2014013456

目录

1 实验简介

本实验中，我们使用Haskell语言完成了抽象语法树的设计、代码解析、实现简单程序的解释执行和编译执行。具体实现情况如下。

1.1 功能实现

功能	难度	是否实现
独立主程序	★	是
REPL	★	是
解释器	★	是
编译器	★★★★	是
文法解析	★★	是
Pretty-printer	★	是
错误处理	★	是
代码测试	★★	是
代码风格	★★★	是
性能优化	★★★	是

1.2 语言特性

功能	难度	是否实现
逻辑表达式	★	是
浮点算术表达式	★	是
字符串与列表	★★	是
While语言	★★★	是
数组	★★★	是
一阶函数	★★★	是
输出语句	不在要求范围	是

2 运行说明

前期准备：下载安装stack以及python3、pyinstaller。

2.1 解释器

1. 在终端进入Runner目录，运行tools\setup.sh 或python tools\setup.py 安装项目。
2. 运行tools\compile.sh 或python tools\compile.py 进行编译。
3. 运行tools\run.sh 或python tools\run.py 运行项目。以使用tools\run.sh为例：
   - 运行tools\run.sh -i <file>，在终端输出file中程序运行结果。
   - 运行tools\run.sh -i <file1> -o <file2>，将file1中程序运行结果输出到file2 中。
   - 运行tools\run.sh -t <file>，在终端输出file中程序的抽象语法树。
   - 运行tools\run.sh -t <file1> -o <file2>，将file1中程序的抽象语法树输出到file2 中。
4. 运行tools\run.sh -repl 进入repl模式。在该模式中，输入:i <program>，解释器给出:i 后的程序的执行结果并输出到stdout；输入:t，输出上一段程序的抽象语法树。如果没有上一段程序，输出Nil。
5. 运行tools\test.sh 执行代码测试。

2.2 编译器

在终端进入Compiler目录，运行tools tools\run.shcompile_and_run.sh ¡file1¿ -o ¡file2¿ 或python tools tools\run.shcompile_and_run.py ¡file1¿ -o ¡file2¿ 编译file1中的程序，将会生成对应的python文件（保存在./dist/file2.py中）和可执行文件（保存在./dist/file2中）。

3 支持的语法特性

解释器、编译器所接受的语言为包含了文档3.1、3.2 中所有特性的语言。

3.1 函数Function

一段完整的程序定义必须是由Function List组成，其中Function List是包含若干个Function 的列表。对于每一个Function，我们规定如下文法：

Function ::= (define (functionName var1 var2 $\cdots$) statement)

其中functionName 为该Function 的函数名字，后面的var1,var2,$\cdots$ 为该函数的每个参数的名字，statement 为函数语句体。对于所有Function，有如下规定：

1. Function 的函数名字区分大小写。
2. 两个Function 如果名字相同但是参数个数不同，则认为是不同的函数。
3. 两个Function 如果名字相同且参数个数相同，则认为第二个函数是第一个函数的重定义，在调用该函数时均使用第二个函数。
4. 任何一个Function 必须有返回值。

3.2 语句体statement

对于语句体statement，除了在文档3.2 中定义的set!、skip、if、while、begin、make-vector、vector-set!、return 以外，我们加入了一条新的语句print。print 语句的定义为

statement ::= (print expression)

其作用为输出表达式expression 求值之后的结果。同时，我们对statement 语句体做出以下说明：

1. 所有语句体按照顺序从函数入口到函数出口一条一条执行。
2. 当在Function 中使用了return 语句之后，该Function 之后所有的语句体都会被跳过，将会直接返回return 语句所返回的值。
3. 在任何一个Function 的语句体中，所有由Function 的参数所定义的变量均为局部变量，在语句体中修改这些变量的值并不会影响到这些变量在外部的值。而除开这些变量以外的所有变量都是全局变量。

3.3 表达式expression

对于表达式expression，其格式与文档3.1、3.2 中所定义的expression 的语法要求一致，同时我们对expression 做出以下规定和说明：

1. 对于一个表达式，其计算顺序从左至右顺序执行。
2. 对于and 运算符和or 运算符，我们实现了短路机制。如果运算符左侧表达式的值已经足够推断表达式的值，则不会再对右侧表达式进行计算。

3.4 变量variable

对于声明的一个变量variable，我们做出以下说明和规定：

1. 变量名必须以字母开头，由数字、字母和下划线组成。
2. 不允许调用未声明的变量。
3. 声明数组变量之后，数组内部未被赋值的部分的值均为undefined。
4. 变量的数据类型为弱类型。

以上为所支持的语法特性。

4 文法分析

文法分析的第一步是按照空格、制表符、换行符等不可见字符作为分隔符，将给定的程序分割成多个字符串。

再将程序分割为多个字符串之后，我们可以发现所需要分析的语言满足LL(1) 文法，所以我们可以通过向前查看一位的方式来分析接下来的动作，判断出接下来要分析的语句的类型。根据我们分析语句的不同类型，建立不同类型的AST，从而完成语法树的构建。

文法分析的部分大体需要分析函数、语句体、表达式、变量四个部分，如果在分析过程中发现无法解析的情况，则汇报编译错误（Compile Error），如括号不匹配、未定义的运算符、表达式格式错误等。

5 解释器

我们首先假设文法分析过程中没有出现编译错误，那么我们便可以从文法分析的步骤中获得一棵AST 树。根据这棵AST 树的形状，针对不同节点，我们可以实现不同的运算机制。大体上我们需要以下几种运算机制：

1. 函数的调用。
2. 局部变量和全局变量的访问与修改。
3. 语句体的执行。
4. 表达式的求值。

为了解决函数调用的问题，我们在实现中维护了一张全局的函数表。每当需要调用函数的时候，我们就去这张表中查找我们所调用的函数是否存在，如果不存在则会引发运行时错误。由于我们所有的语句都是在函数内部执行，所以存在局部变量和全局变量的问题。在执行语句体和计算表达式的值的时候，我们始终维护一张全局变量表和一张局部变量表，用于储存每个变量当前的值。当我们访问和修改一个变量的时候，永远优先访问局部变量表的值，如果在局部变量表中没有找到再去访问全局变量表。每次调用函数的时候，重新为该函数创建一张局部变量表，当前的局部变量表重新储存进栈中，等完成函数调用之后再重新取出该局部变量表。

现在我们假设在之前的文法分析的过程中可能出现编译错误，对于-i 运行模式和-t 的语法树模式，我们只要直接汇报对应的编译错误即可。但是如果实在repl 模式下，我们并不希望因为编译错误或者之后解释过程中的运行时错误就导致退出repl 模式，所以在repl 模式中我们加入了错误侦测，用于保证即使出错也不会退出repl 模式。

在解释器执行的过程中如果发生错误则会导致运行时错误（Runtime Error），如访问未定义变量、函数不存在、数组越界、类型不匹配等错误。

6 编译器

由于所给的文法是弱类型的文法，比如函数、参数等都没有类型说明，翻译到强类型的比如LLVM IR这样的中间语言会引入一些新的问题，虽然理论上可以通过cast强制类型转换来做到这一点，不过这意味着所有变量都会先声明64位地址，造成内存空间浪费。参考一些已知的编译器如bumba将python编译为ir，但他们在语法上要求在函数的装饰器里声明类型。所以我们先选择用python3作为目标语言。

在翻译的过程中，我们遍历了两遍语法树：

第一遍找出所有的变量声明结点，并将他们保存在globalVariable列表中（这样做的原因是我们认为文法中声明的所有变量都是全局变量，只有传递的参数是局部的，所以在python中需要在外部声明这些全局变量，并在函数中标注global <variable name>）。

第二遍开始自上而下的翻译，主要是将文法中的前缀表达式变为中缀表达式。同时在表达式求值的过程中，我们实现了constant propagation，即操作符两边是常数时直接给出运算结果。

最后，我们利用pyinstaller这个外部工具生成可执行文件，这一步通过在Translator.hs中通过System.Process直接调用自动执行。

7 测试及测试结果

7.1 关于正确性的测试

7.1.1 针对预定代码的测试

正确性第一部分的测试为利用test 模块对写好的代码进行运行，比对运行的输出和程序应有的预期结果。在test\test_file 中，提供了所有被编译解释语言的代码：

- arr：为对数组使用的测试。
- fib,fib_arr：为斐波那契数列的的两个测试用例。
- func1,fcun2：为对函数使用的测试。
- middle：二维加权重心的测试用例。
- qsort,qsort_big：为两个快速排序的测试用例。
- queen,queen_fast：为两个八皇后的测试用例，其中queen_fast为加速过后的版本。
- test1,test2,test3：为三个基本的语法测试用例。

在每个文件夹下有三个文件：code、answer、output，分别代表：

- code：被检查的源代码。
- answer：代码应有的输出。
- output：代码实际运行的输出。

在执行正确性测试的时候，会依次针对每一个代码做正确性检查。

7.1.2 针对属性的测试

利用Haskell 的QuickCheck 功能，正确性检查的第二部分会针对以下三个属性进行测试：

1. prop_add：常量的加法运算：每次测试会生成两个数，根据这两个数生成对应的加法表达式，计算其结果应该与直接使用Haskell 的加法运算结果一致，每次测试100 组数据。

2. prop_qsort：每次随机生成一个列表，同时生成关于这个列表进行快速排序的代码，并利用解释器执行得到排序之后的结果。该结果应该与直接使用List.sort 的结果一致，每次测试100 组数据。

3. prop_queen：每次随机生成一个整数n，同时生成一段求n 皇后方案数的代码，并利用解释器得到其结果。该结果应该与n 皇后的实际方案数一致，每次测试20 组数据。

7.2 测试结果

7.2.1 正确性

以上所有测试通过解释器和编译器的测试结果均与answer文件相吻合（但在浮点数精度上两者的运行结果有差异）。

7.2.2 性能优化

- 短路：在and 表达式和or 表达式中，如果表达式的左部表达式的值已经足够决定该表达式的值，我们就不会再计算右部表达式的值。这样可以将and 和or 表达式的计算速度提升恰好一倍。

- Constant Propagation：

  我们用编译器生成表达式(*print* (− (+ 3 5) 4))的可执行文件，并测试运行时间来对这一性能进行比较，以下测试时间由Linux的time命令给出。

	优化前	优化后
real	55ms	32ms
user	14ms	11ms
sys	16ms	10ms

8 实验体会

本次实验中，我们将课上学到的知识付诸实践，完成了抽象语法树的设计、代码解析、实现简单程序的解释执行和编译执行等功能。在这个过程中，我们再次复习理解了Haskell，对这门函数式编程语言有了更深刻的理解。

巧合的是，在本学期的另一门课程计算机与网络体系结构（2）中，我们也实现了一个简单的编译器。通过这两个目标相似的实验的对比，我们更体会到了Haskell相较于其他命令式语言的独特之处。

另外一点，我们实现解释、编译的语言是一门非函数式的语言（有变量的赋值，以及我们引入的输出语句），而Haskell 是一门函数式的语言，这也充分让我们理解到了函数式语言和非函数式语言在各种方面的不同，以及它们实际上是完全等价的这一特点。