《函数式语言程序设计》课程作业

邵成, 程玺, 周旻

2016 年 12 月 30 日

1 简介

本次作业选题为“编程语言的解释器/编译器的实现”。本提纲列举一系列编程语言特性的语法和语义，学生可选择其中一部分，使用Haskell进行实现其解释器或者编译器。本作业主要目的为鼓励学生使用Haskell语言完成抽象语法树的设计、代码解析并实现简单程序的解释执行或者编译执行。

1.1 完成方式

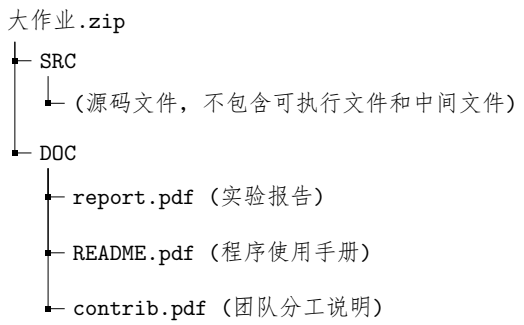
本作业允许组队完成，规模不得超过3人。作业提交内容包括：

1. 项目全部源代码；
2. 实验报告（包括但不限于：（1）提交代码的执行方法描述，（2）作业的实现目标（实现了哪些语言特性、语言定义是怎样的、实现了哪些得分项），（3）作业的实现思路与亮点，（4）挑战与解决方案、参考资料或代码等）；
3. 每位成员在作业中的贡献（请用一个另外的文档简要说明）

作业文件应打包成`.zip`或`.tar`格式，请勿用快压等软件创建压缩包，以免出现压缩包不能正常解压的问题。作业文件的名称应为如下格式：

`[姓名]_[姓名]_大作业.zip`

其中文件名应包括团队所有成员的姓名并用下划线分隔开。压缩包内的组织结构应如下：



本作业的截止时间为**第18周周日(即2017年1月15日)晚23:59**，请按时提交作业，否则会影响成绩的录入。

1.2 评测方式

本作业的评测方法为白盒+手动测试。即通过以下方面进行详细评分：

1. 提交的实现首先会经过基本的测试用例测试，
2. 提交的实现会经过边界值等特殊情况测试，
3. 所提交实现的性能优化和代码风格等均会考察和予以评分。

执行环境为 GHC 8.0.1，允许使用所有 GHC 扩展和 ghc boot libraries（GHC自带库）。鼓励学生使用 `cabal` / `stack` 进行项目管理，使用第三方库需要提前向助教确认并获得批准。

本作业要求每个小组在语言特性列表中选定一部分特性，设计一个Haskell类型表示其抽象语法树，实现一个将该类型解释执行到最终结果的函数。

每个作业的必选功能总难度为4颗星，必须完成的语言特性总难度为2颗星。对于多选的语言特性与功能实现项目，在本作业成绩上将有加成。对于本作业成绩超过满分者，超出部分可累积至本学期总成绩。

2 功能实现得分项

2.1 执行方式

2.1.1 独立主程序

在文法解析器的基础上，实现可独立执行的主程序。主程序可以通过

表 1: 功能实现项目的列表

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

解析命令行参数，获得程序路径和输出路径，读取程序并将执行结果输出到指定位置。

2.1.2 REPL

在实现独立主程序的基础上，实现REPL（Read-Eval-Print-Loop），按行接收输入程序并即时给出执行结果。

2.2 解释 / 编译

2.2.1 解释器

实现解释执行功能。入口函数为`eval`。

2.2.2 编译器

除解释函数以外，将程序翻译到中间语言(如`LLVM IR`，`GCC GIMPLE`)或特定指令集下的汇编语言(如x86 Assembly)，通过现有工具将中间语言或汇编语言转换成二进制可执行文件。实现简单的编译优化操作（如constant propagation等）有额外加分。推荐结合“代码测试”和“性能优化”实现，用解释器实现测试编译器实现的正确性，并对比二者性能，会有额外加分。实现该项功能点请在实验报告中注明可执行文件的目标平台。

2.3 解析与显示

2.3.1 文法解析

实现具体文法的解析器（`parse`函数），从字符串解析到抽象语法树。语言特性列表中所有特性的文法均基于S表达式[1]设计，不含左递归构造，比较简单，方便使用parsing组合子进行解析。

2.3.2 Pretty-printer

实现Pretty-printer用于将解析获得的抽象语法树格式化输出成可读的字符串形式。Pretty-printer应实现为`Show`的实例。只要能够将抽象语法树输出出来就算基本完成了该项功能，但如果要拿到较高分数那么输出的语法树需有较好的可读性。关于输出格式的设计可以参考`Clang`输出的C语法树。

2.4 代码规范性

2.4.1 错误处理

在解析器和解释器实现中，处理可能出现的错误情况（如类型不匹配、显式抛出异常等），并返回有一定信息量的错误信息。

2.4.2 代码测试

实现一系列测例和单元测试代码，用于测试解释器的正确性。使用`QuickCheck`等支持随机生成测例的框架，有额外加分。

2.4.3 代码风格

鼓励使用`Monad`、`Monad Transformer`、`mtl`等抽象工具组织代码，实现更简练的解释器。如果代码书写方式满足以下要求，有额外加分：

- 可组合（composable）：对特定的操作（解析、解释、pretty-printing等），不同特性集上的实现可以放在不同module中分离编译，并组合起来形成组合语言的实现。

[1] 具体文法请参考章节3

- 可扩展（extensible）：对特定的特性集，不同操作的实现可以放在不同module中分离编译，并方便在新module中实现新的操作。

2.4.4 性能优化

通过使用适当的数据结构等方式改善性能。实现对比不同实现方案测量运行时间的benchmark，有额外加分。

3 语言特性得分项

表 2: 语言特性项目的列表

语言特性	难度	必选
逻辑表达式	★	是
浮点算数表达式	★	是
字符串与列表	★★	
While语言	★★★	
数组	★★★	
一阶函数	★★★	
文法作用域	★★★	
高阶函数	★★★★	

3.1 表达式语言

3.1.1 逻辑表达式

难度：★

文法：

```
expression ::= True | False
    | (not expression)
    | (and expression expression)
    | (or expression expression)
```

介绍：

实现布尔值常量和and/or/not运算。

3.1.2 浮点算术表达式

难度：★

文法：

```
number ::= integer | integer.integer

expression ::= number
    | (+ expression expression)
    | (- expression expression)
    | (* expression expression)
    | (/ expression expression)

    | (= expression expression)
    | (< expression expression)
    | (<= expression expression)
    | (> expression expression)
    | (>= expression expression)
```

介绍：

实现浮点数常量和四则运算。若同时实现了“逻辑表达式”特性，可实现浮点算术表达式上的比较操作，返回布尔值。

为简化实现，浮点数常量的具体文法仅包括十进制整数或小数点连接的十进制整数，无需支持负数常量；无需实现“取反”的单目运算符。允许在运算过程中出现`Infinity`/`NaN`。

和“逻辑表达式”特性一同实现时，解释函数的返回类型至少需要包含`Bool`/`Double`两种case（例如`Either Bool Double`），同时考虑到解释过程可能出现的类型不匹配的错误。

3.1.3 字符串与列表

难度：★★

文法：

```
expression ::= nil
    | (cons expression expression)
    | (car expression)
```

```
    | (cdr expression)

    | char_literal
    | string_literal
```

介绍：

实现列表类型，并基于列表类型实现字符串。cons将两个子表达式值组成一个pair，car将pair的左侧值取出，cdr将pair的右侧值取出。nil则代表空列表，与cons组合使用，即可构造出列表。另外，需要实现字符常量和字符串常量；字符常量使用英文半角单引号（如'a'），字符串常量使用英文半角双引号（如''hello world''），实际转化到字符列表实现。

更复杂的数据结构（如二叉树等），同样可以基于pair进行实现。

3.2 命令式语言

3.2.1 While语言

难度：★★★

文法：

```
expression ::=
      normal_expression[2]
    | variable

statement ::= statement_list
    | (set! variable expression)
    | skip
    | (if expression statement statement)
    | (while expression statement)

statement_list ::=
    | (begin statement ..)

program ::= statement
```

[2]本节出现的normal_expression指的均是3.1节定义的基本表达式

介绍：

实现一门有副作用（对变量赋值）、带一定控制流特性的语言，至少需要先实现“逻辑表达式”特性。解释执行的过程需要顺序执行`statement`列表，并维护从变量名到值的映射。赋值操作会新增变量或修改已有变量的值，而尝试读取不存在的变量的值将导致运行时异常。

3.2.2 数组

难度：★★★

文法：

```
expression ::=
      normal_expression
    | variable
    | (vector-ref variable expression)

statement ::=
      (make-vector variable expression)
    | (vector-set! variable expression expression)
```

介绍：

基于While语言，加入数组特性。`make-vector`给定变量名和数组长度，新增一个数组变量；`vector-ref`给定数组变量名和下标，读出该下标指向的元素；`vector-set!`给定数组变量名、下标和值，对数组指定位置进行赋值。

运行过程中，要求下标为非负整数，且不得超过新建数组时指定的数组长度。数组下标从0开始。

3.2.3 一阶函数

难度：★★★

文法：

```
expression ::=
      normal_expression
    | variable
    | (function-name expression ..)
```

```
statement ::=
      (return expression)

function ::=
      (define (function-name variable ..) statement)

program ::=
      function ..
```

介绍：

基于While语言，加入定义一阶函数的特性。表达式新增“函数应用”种类；增加“从函数体返回”的statement；程序体由1个或多个函数定义构成，执行过程从main函数开始。

一阶函数特性不支持传入函数作为参数或返回函数，实现难度小于“高阶函数”。但是需要支持递归调用，并妥善处理作用域的问题；表达式的求值过程可能带有副作用，因此规定在函数应用表达式中，各参数表达式的求值顺序从左到右。传入参数数目与定义函数时参数数目不同时，视为运行错误。

3.3 函数式语言

本节语言特性不推荐与While语言系列的语言特性协同实现。

3.3.1 文法作用域

难度：★★★

文法：

```
expression ::=
      normal_expression[3]
    | variable
    | (let variable expression expression)
```

[3]本节中的normal_expression均指3.1节定义的基本表达式或3.2节中拓展后的表达式(取决于你实现出的语言特性)

介绍：

实现“文法作用域”（Lexical Scoping），允许将表达式的值绑定到变量名，表达式求值过程需要维护从变量名映射到值的作用域。

3.3.2 高阶函数

难度：★★★★

文法：

```
expression ::=
      normal_expression
    | variable
    | (lambda variable expression)
    | (expression expression)
```

介绍：

实现无类型 λ 演算，支持定义和使用匿名函数。

4 实现要求

作业源码应能编译成一个解释器可执行文件(必选)以及一个编译器可执行文件(可选)。解释器和编译器的文件名分别为`ki`和`kc`(在Windows系统中它们会带上`.exe`的拓展名)。我们对它们的用法做一些最小规定(规定外的其它用法请在`README`中注明)。

4.1 解释器

4.1.1 解释执行

用法： `ki -i <file> -o <file>`

解释： `-i`指定输入文件的路径。`-o`指定将解释的结果输出到哪个文件。如果不指定`-o`选项，那么结果应输出到`stdout`。

用法： `ki -t <file> -o <file>`

解释： `-t`后跟程序文件的路径。该命令输出指定程序的抽象语法树指定文件或`stdout`。

4.1.2 REPL

用法： `ki -repl`

解释： 进入REPL模式，该模式下用户输入一段指令，回车后解释器输出执行结果到`stdout`，并等待处理用户的下一条指令。

用法： `:i <program>`

解释： 在REPL模式下，解释器给出`:i`后的程序的执行结果并输出到`stdout`。

用法： `:q`

解释： 在REPL模式下，该指令退出解释器程序。

用法： `:t`

解释： 在REPL模式下，该指令输出上一段程序的抽象语法树。如果没有上一段程序，应输出错误信息或不输出。

4.2 编译器

用法： `kc <file> -o <file>`

解释： 第一个参数为输入程序的路径，`-o`指定输出的可执行文件的路径。默认地该可执行文件的目标平台应为当前平台。

用法： `kc <file> -o <file> -is <arch>`

解释： 如果你的编译器支持输出到多种平台上的可执行文件(比如你的编译器将程序翻译为`LLVM IR`)，那么`-is`参数指定生成可执行文件的输出平台，`<arch>`中的内容请参考`Clang`的`-target`参数[4]。

4.3 其它说明

1. 请在实验报告的首页注明你们团队完成的功能项目和语言特性。我们会据此检查作业完成的情况。

2. 如果你实现了一阶函数这一功能点，那么程序的入口函数永远是`main`。否则，如果一个程序文件中包含有多个表达式，那么程序的执行结果应为顺序解释这些表达式产生的结果。

3. 我们不考虑一个程序由多个文件组成这一情形，即每个程序都是单文件的。如果你的作业支持多文件程序的处理，那么请在`README`中说明

[4] `http://clang.llvm.org/docs/CrossCompilation.html`

相应解释器和编译器的用法，我们也会提供额外的加分奖励。

4. 如果你完成了性能优化功能点，请在实验报告中通过足够的实验数据以及分析来充分讨论不同实现方法、不同优化技术的优缺点。