

编译原理实验报告

Lab1 Lexical Analyzer Programming

Lab2 Syntax Parser Programming

学号： 57120201 姓名： 胡文韬

二0二三年二月

### 编程语言简介，环境配置，参考，计划

**语言简介：**Haskell是一门函数式语言，同Lisp和Schema一道作为第四代图灵完全语言的代表作，比matlab和python初始功能更强大，语法也更严谨，作为编译器大神圈开发的小玩具，支持梦幻般的模式匹配，用它来编写parser优雅而简练。

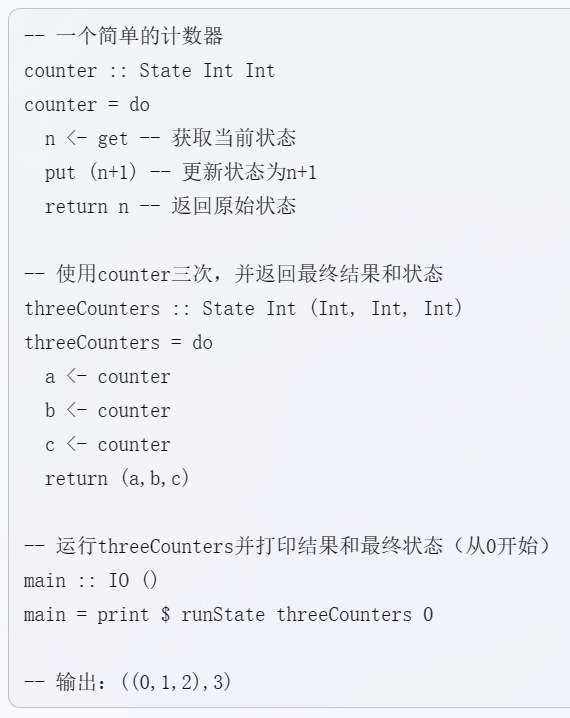
**语言核心State Monad：**[简单来说，state monad是一种在haskell中模拟有状态代码的技术](https://acm.wustl.edu/functional/state-monad.php" \t "https://www.bing.com/_blank)[。可以通过一系列函数调用，将一个状态变量（可以是任意复杂的类型）传递下去，并根据状态变量产生不同的结果](https://wiki.haskell.org/Monad" \t "https://www.bing.com/_blank)。

state monad的定义如下：

**newtype State s a = State { runState :: (s -> (a,s)) }**

[这个定义表示，state monad是包装了一个函数的haskell内置类，这个函数接受一个状态s，并返回一个值a和一个新的状态s](https://acm.wustl.edu/functional/state-monad.php" \t "https://www.bing.com/_blank)。

[Monad类可用于实现state monad。可以使用do-notation和bind操作符（>>=）来组合不同的有状态操作](https://wiki.haskell.org/Monad" \t "https://www.bing.com/_blank)。例如（摘自haskell社区与github）：

****

**Motivation：**我热爱Haskell语言久矣，用它来写编译器被haskell社区称为入门项目，所以此次实验并不拘泥于课程要求只写parser，还完成了语法制导翻译，至于优化，尚未熟悉LLVM的使用，加之同haskell版本不兼容问题比较严重，暂时放弃。

**编译器：**GHC-7.2.2

**参考：**Monadic Parser Combinators，How to build a monadic interpreter in one day 以及haskell社区提供的其它论文与代码资料。

使用即时交互GHCi（语法上和GHC有一定区别）

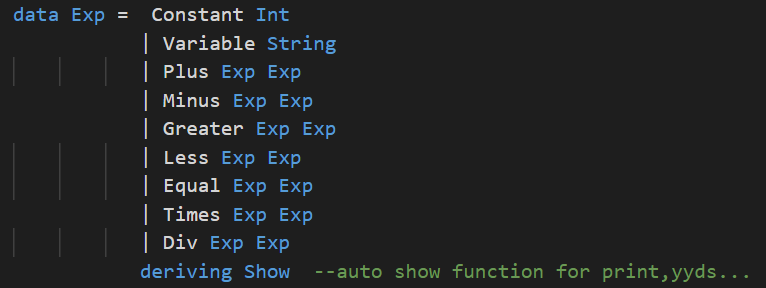
**计划：**先写后端，因为AST作为输入足够提供语义。再写前端parser，用Monadic Parser Combinators的自底向上结合各RE->NFAs，monad会同时提供下推自动机LL(1)的状态转换，用自动模式匹配作为下推动力。对于错误检查，不符合语法即报错。

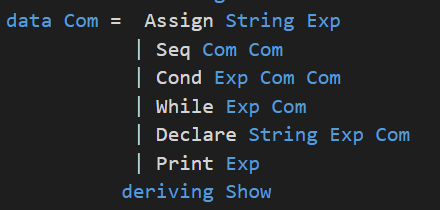
### **自定义语言和符号简述**

自定义一种小型函数式语言。

数据类型：常量，变量（String/带符号整型）

操作符：支持四则运算比较和阶乘，四种卫式命令：连接（；），赋值（Declare），条件（Condition）与循环（While），此外，支持排序（Seq），打印（Print）以及文件读写式交互。

****

****

**语法使用和语言示例：**

**declare x = 150 in**

**declare y = 200 in**

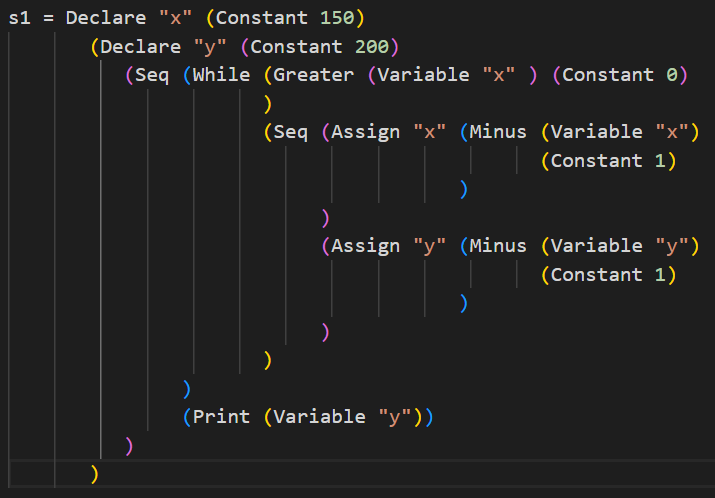
**{while x > 0 do { x:=x-­1; y:=y-­1 };**

**print y**

**}**

**该程序输出结果是50**

**前端输出/后端输入AST示例：**

****

**即：用括号说明归约顺序，同时标出数据类型：变量和常量。**

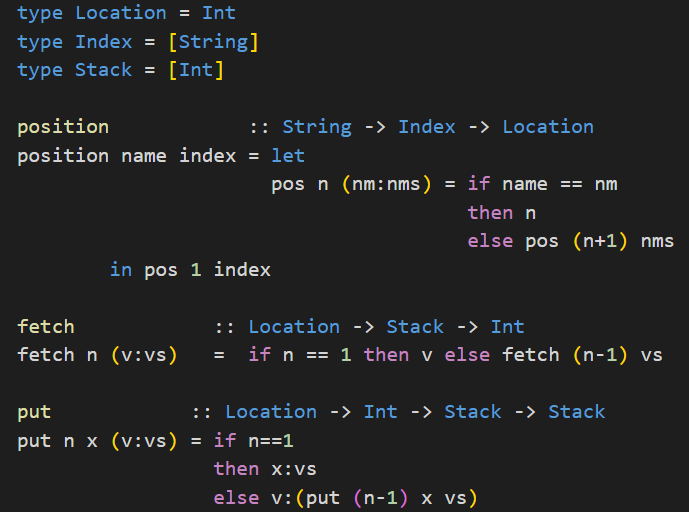
### **实现方法概述与代码展示**

1. **后端**

**常量：**不给标识符，也不单独存在常量表里，直接用语言本身的常量表。

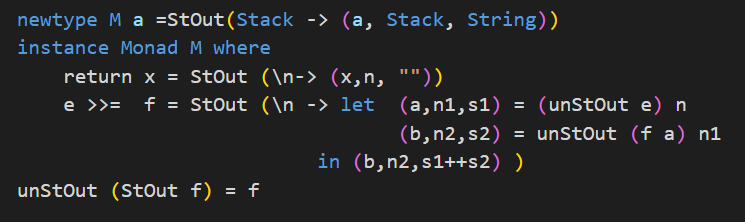
**下推自动机模型SLR(1)的实现：**

**存储：**用location作为索引存储在栈stack里，字典[Index]即变量表，结合起来也可以写为[(String, Value) ]，分散写法中，position函数用于使两个数组表中标识符及其值相对应



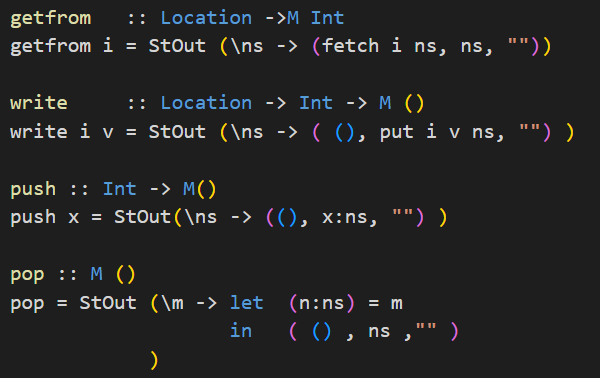
**下推动作：**用haskell的自动递归+模式匹配，fetch取用栈顶元素，put函数重写更新后的环境：新的双栈根据更新的位置、更新的存储值和先前内容来计算。

**StOut：**“State and Output” Monad. 将一个stack映射到一个元组，其中a是对当前栈的状态等进行计算的输出结果返回值，第二个Stack是状态更新后的栈，第三个是该值的标识符。

**UnStOut：**从匹配的Monadic capture中提取内嵌的内容（函数即产生式或标识符）

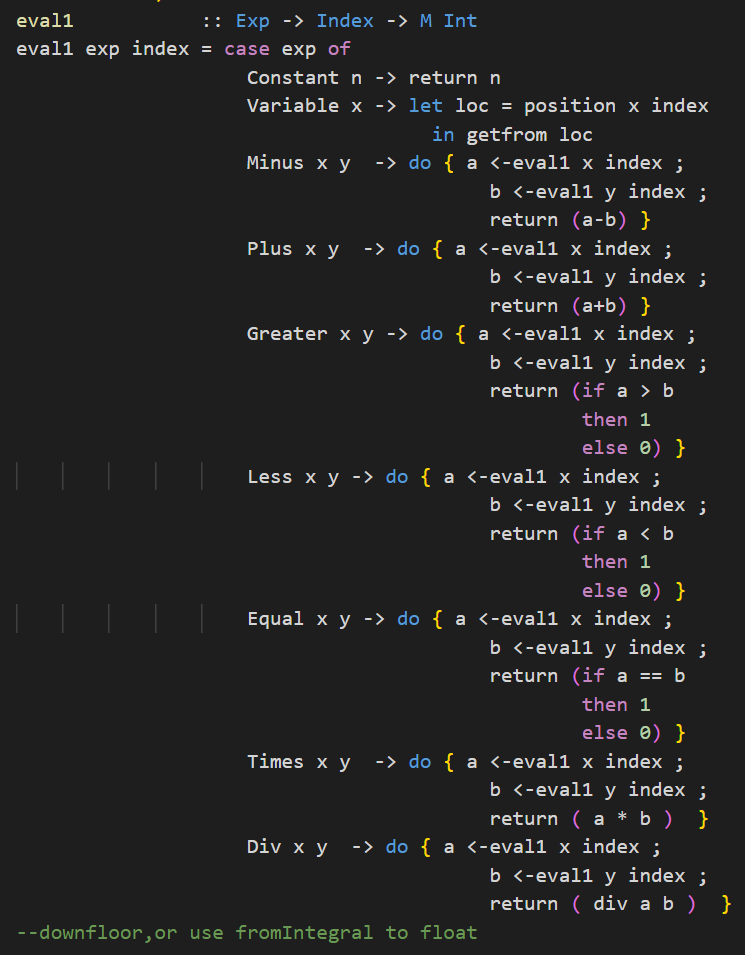
Getform从栈中取出计算结果，write将计算后的变量及其值放在栈顶。

Push通过将值推为栈的顶部来修改栈（而不是索引字典）,可用于伴随读取AST树创建一些新变量，必须伴随着新标识符插入索引。Pop进行出栈返回值到栈顶的操作。

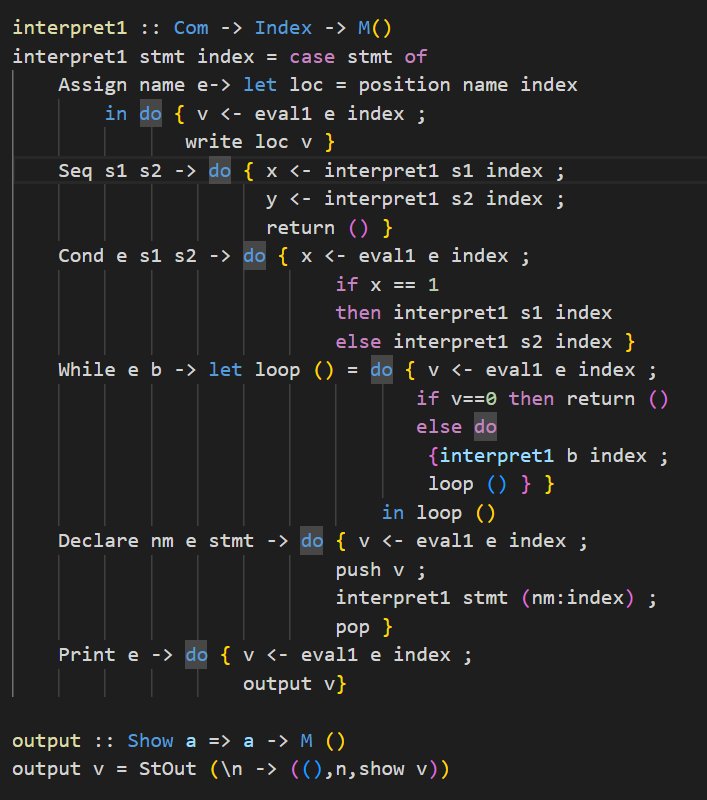


以上短短几行代码已经组成了语法制导翻译的下推自动机的完整架构。下面是语义操作：

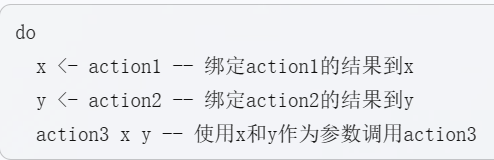
操作符翻译：



命令符翻译：



\*\*\*[do-notation是一种在haskell中编写monadic代码的语法糖](https://www.educba.com/haskell-do-notation/" \t "https://www.bing.com/_blank)，[可以让我们用类似于命令式语言的方式来组合不同的monadic操作，如输入输出、状态变化、异常处理等](https://www.educba.com/haskell-do-notation/" \t "https://www.bing.com/_blank)，基本形式：



Output函数实现结果打印输出的工作。

1. **前端（parser）**

正规文法产生式如下：

数值运算：

**<rexp> ::= <rexp> <relop> <expr> | <expr>**

**<expr> ::= <expr> <addop> <term> | <term>**

**<term> ::= <term> <mulop> <factor> |< factor>**

**<factor> ::= <var> | <digiti> | ( <expr> )**

**<var> ::= <Identifier>**

**<digiti> ::= <digit> | <digit> <digiti>**

**<digit> ::= 0 | 1 | ... | 9**

**<addop> ::= + | -**

**<mulop> ::= \* | /**

**<relop> ::= > | < | =**

命令：

**<com> ::= <assign> | <seqv> | <cond> | <while> | <declare> | <printe>**

**<assign> ::= <identif> ":=" <rexp>**

**<seqv> ::= "{" <com> ";" <com> "}"**

**<cond> ::= "if" <rexp> "then" <com> "else" <com>**

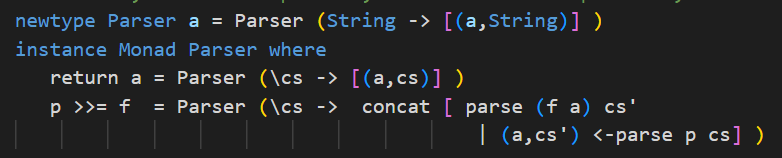
**<while> ::= "while" <rexp> "do" <com>**

**<declare> ::= "declare" <identif> "=" <rexp> "in" <com>**

**<printe> ::= "print" <rexp>**

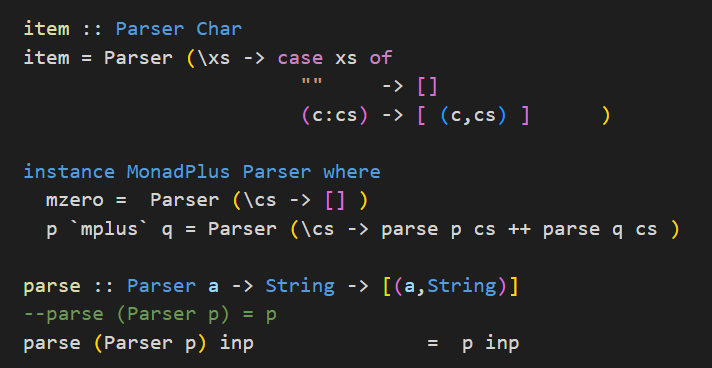
下面展示parser的实现：用课堂讲的自底向上机器实现法，先实现小的单独的子式，再组合起来实现复杂功能。

先定义parser的一般形式作为模板类：

****

每个parser实际上都是从字符串到字符+字符串的列表（下推自动机的栈）的函数。

同时，每个parser都是一个小型自动机（FA）

****

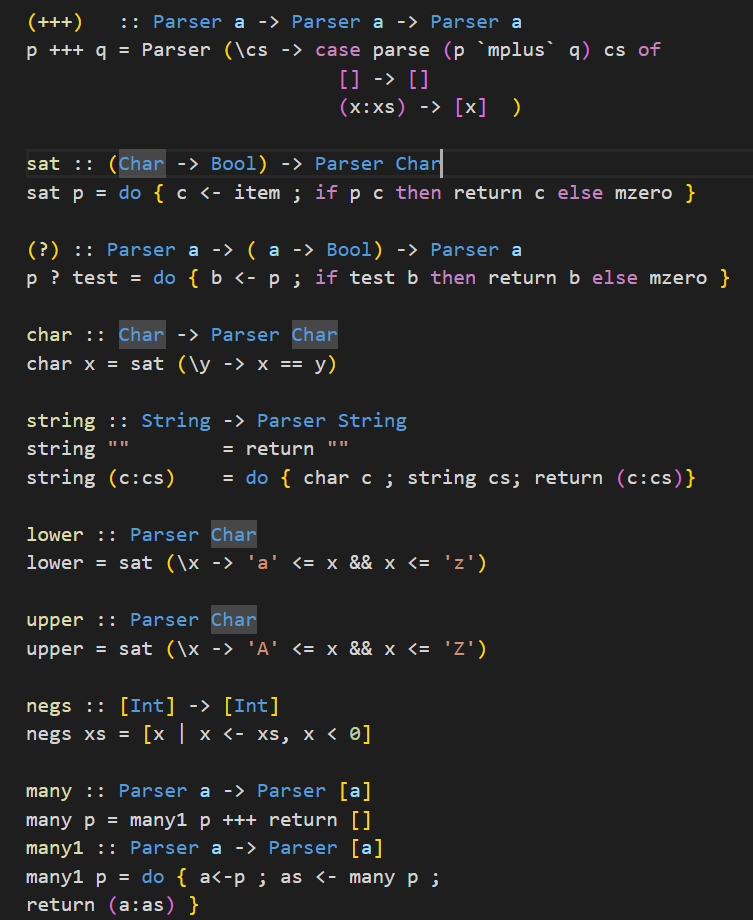
然后定义parse从输入取用字符的动作，小自动机combine动作与终止匹配（待parse的字符串为空），用+++这个隐函子把它们封装在一起，用many把输入字符处理为状态，匹配关键字的思路是组合所有parser的结果并提取有定义的小parser的运行结果。

sat或(?)隐函子帮助创建一个遵循特定规则的parser，把匹配的操作符改写为成函数式的标准前缀写法，而失配的字符则处理为空字符。

Char和String定义了识别对应于这两种数据类型的parser。

Lower和upper框定了输入出现的字母范围。

Negs从输入数字串中提取负数。



下面介绍从输入串去除空格的parser:

（C++和java可以在读文件时自动不读取空格，但haskell就需要调库实现，本实验我没有使用parser的相关内置库，而是自己手写一遍逻辑）

Sapce，结合sat识别空格换行符制表符等

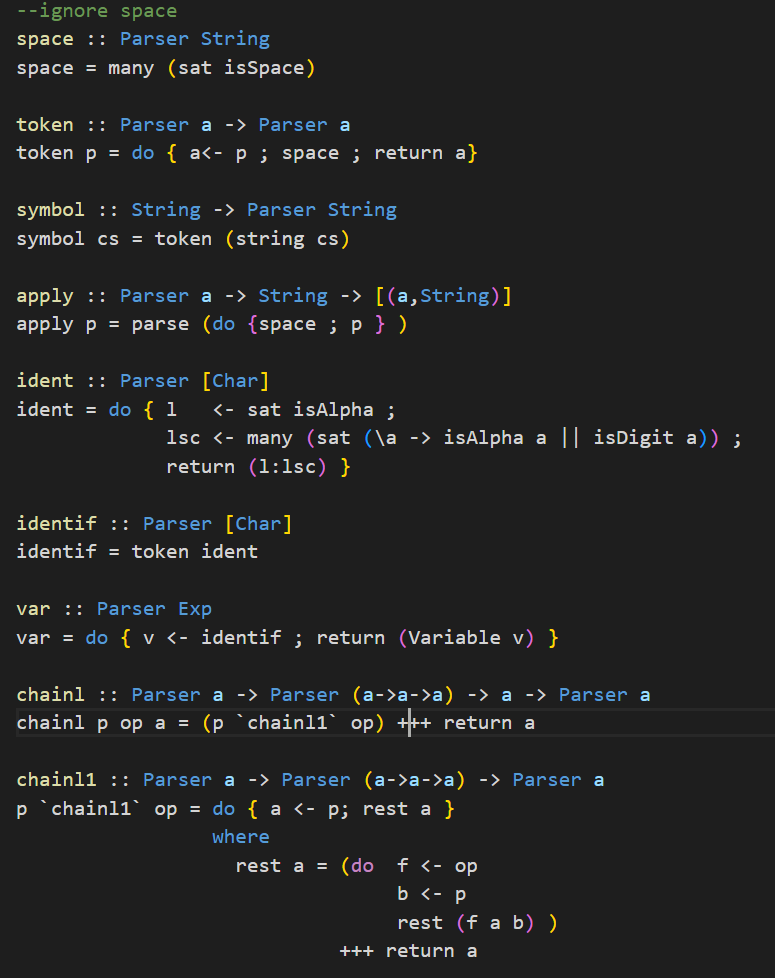
Token，将空格后的标识符识别为token

Symbol，将空格前后的特殊关键字（操作符，string类型）读取为token

Apply，从输入串开始消除空格（调用parse动作）

apply :: Parser a ­> String ­> [(a,String)]

apply p = parse (do {space ; p } )



下面介绍变量标识符的parser：

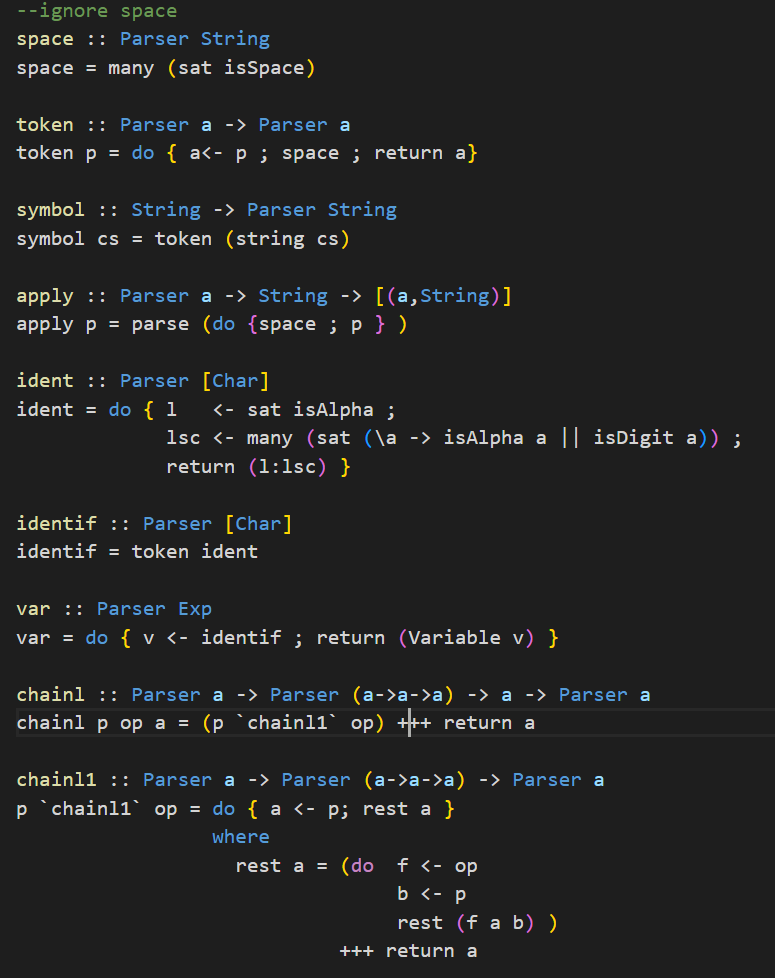
ident调用内置库自动识别输入是字母还是数字

Identif对付如果标识符之间连有空格的情况

Var识别变量标识符并返回识别结果（以括号+variable v的形式）

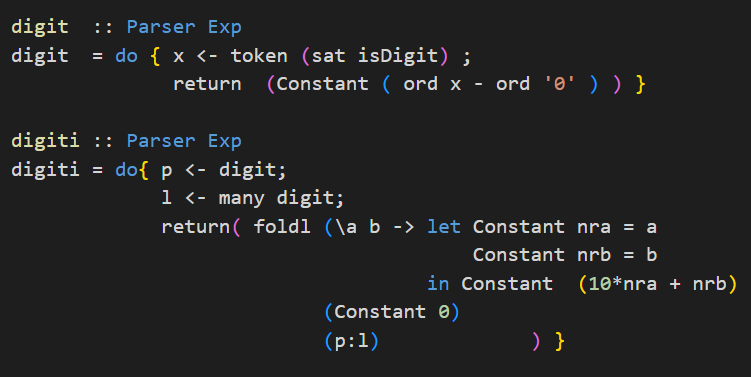
Chainl是组合标识符和原子操作符的parser,chainl1是组合两种原子操作符的parser

重复调用两个parser组合起所有原子操作符便能得到整个语言（ASTs的形式）

****

**下面展示原子操作符和标识符的parser：**

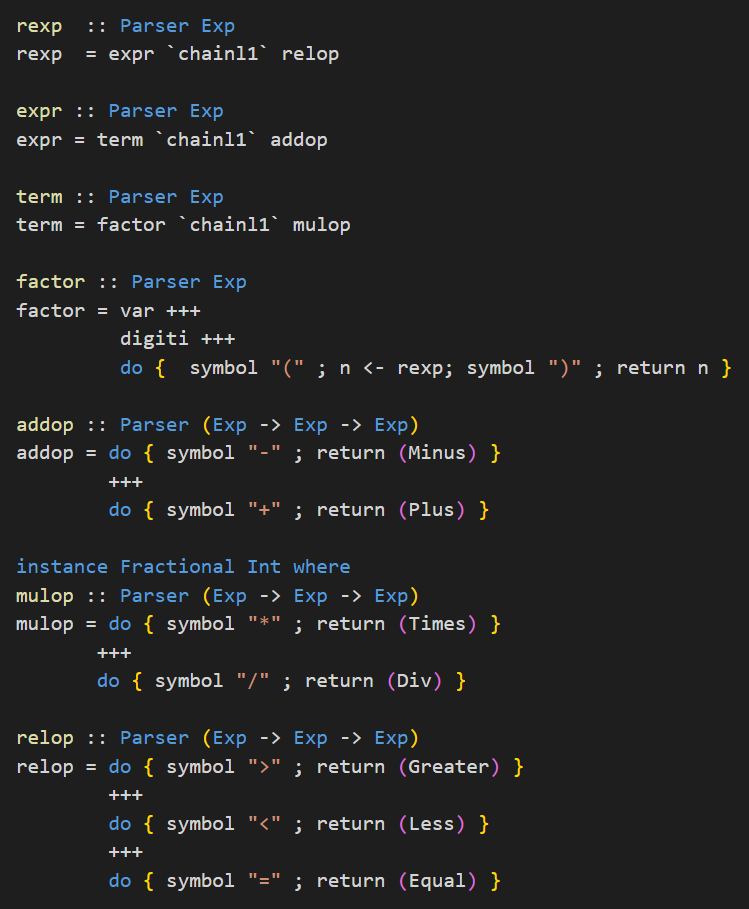
**数字可以直接写（做一下连续输入的十进制数值计算）**

****

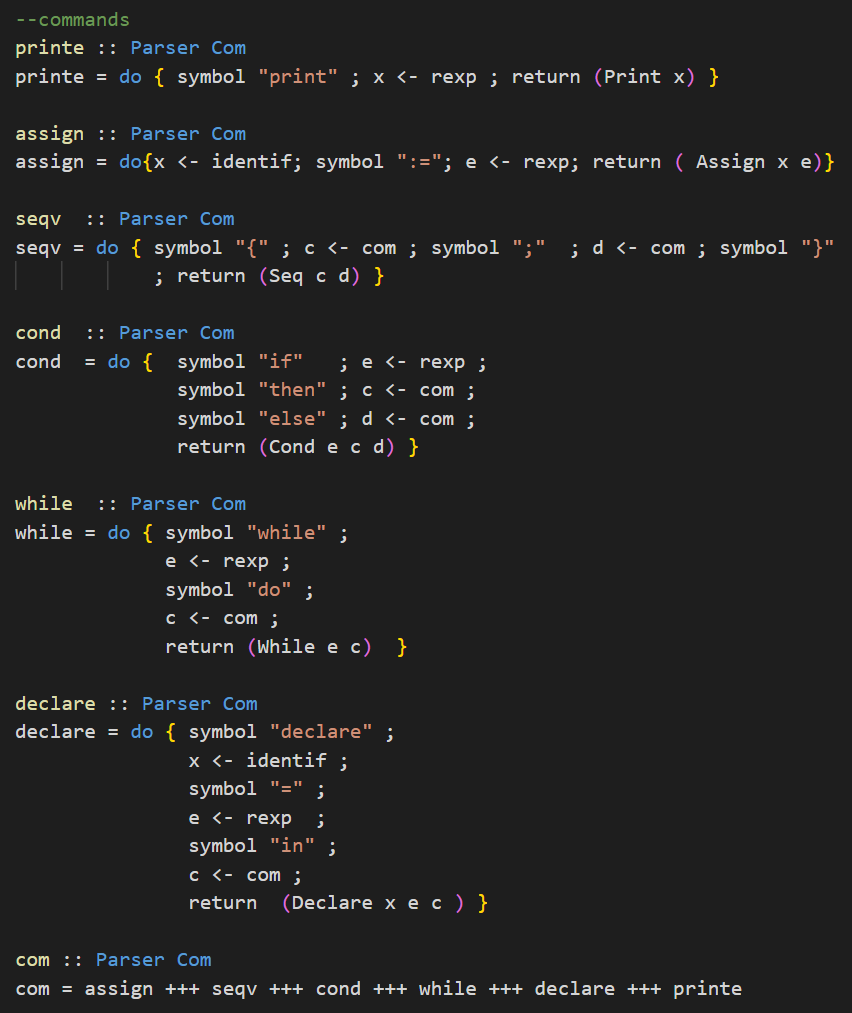
**表达式需要chainl1做转接，以及按照优先级进行识别（归约）与组合：**

**+- addop的优先级小于\*/ mulop小于比较 relop小于括号（以及阶乘factor）**

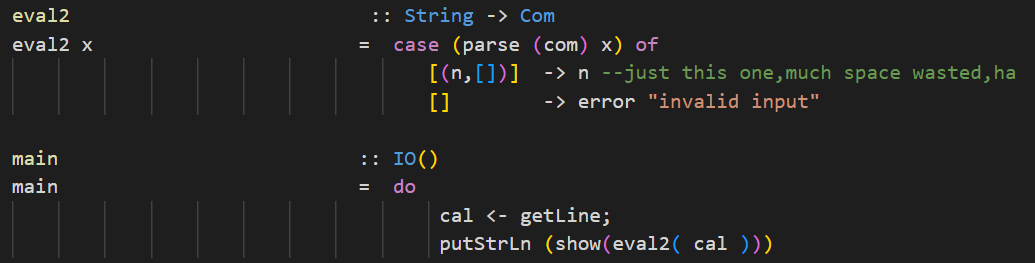
**命名与规约顺序在正则产生式中已经定好，下图可以看出我的parser写法和自然表达式的写法十分接近，这也是haskell的优美之处。**

****

**命令也一样，十分自然：**

****

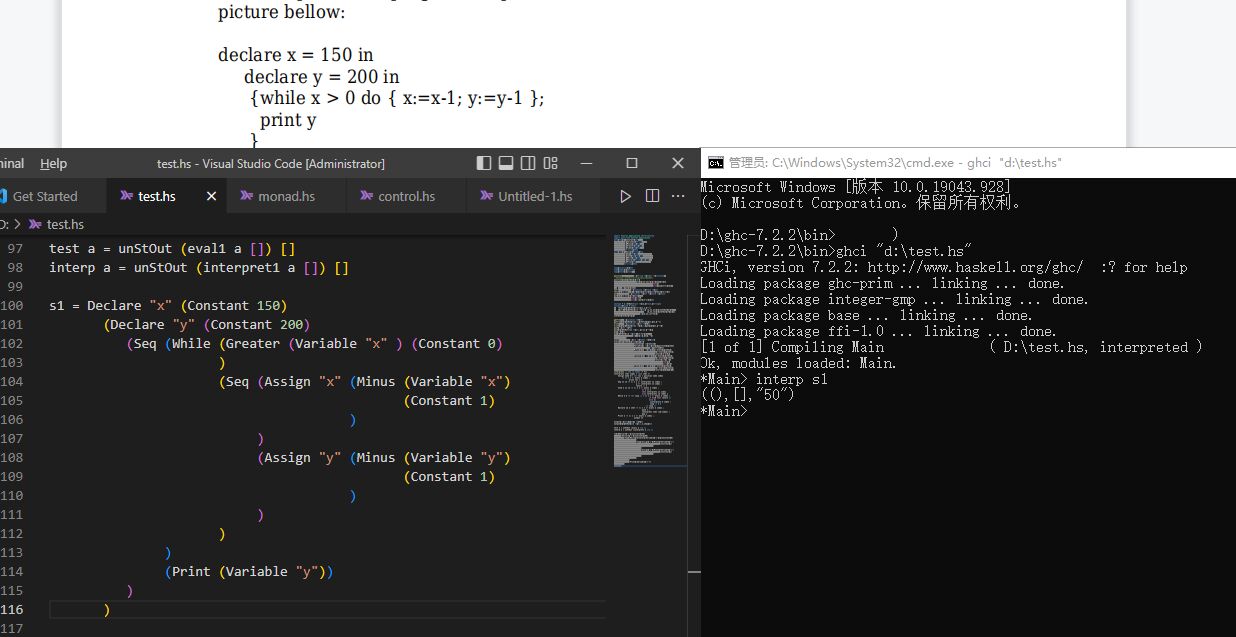
**测试与输出：**

****

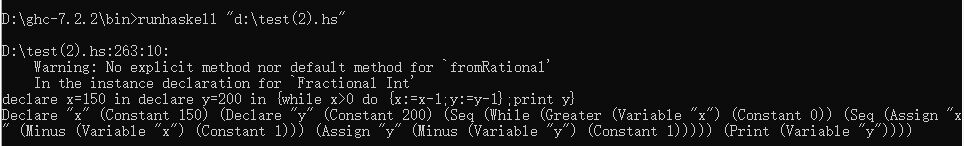
**报错处理写的简单，只要有一个小FA没有匹配成功就终止匹配，输出空栈，所以parse的结果是空list对应报错。如果匹配到最后成功，状态栈应为空(S出栈)，数值栈是后端执行结果，就把该结果从栈中取出并输出。**

### **实验结果展示**

**（GHCi交互模式）下图为后端测试结果：输入AST树输出执行结果：**



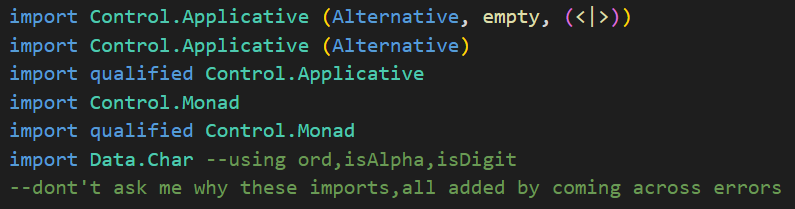
**下图为前端测试结果，输入自定义语言的源代码输出AST树（括号说明规约顺序）**



**（助教如果要验证的话可能需要配置GHCi环境，可以联系我拍视频等等。输入输出文件的示例仅供参考，我按照教程加入GHC格式的文件读写报错并放弃。最后命令行测试时后端不参与测试）**

### **实验感想**

**因为有Hutton大神的Monadic Parser Combinators(1998)，所以前端写得非常顺畅，但monadic类的不熟悉以及GHC版本更迭的缘故，部分语法不再支持，部分内置库脱出弃置，更换了较多写法（尤其是bind操作），回头看前面杂乱琐碎的import，都是github上一点点查issue一点点加上与改写的辛酸。**



**但haskell这门函数式语言依旧是我最爱的语言，优雅简洁，贴近自然语言。在github上看到很多国外的大学（如东京大学）系统学习它，课设大作业是用它写完整的编译器，想起大二上学期软件工程学院的周晓宇老师为我介绍它时说monad类，函子（functor）是对parser进行数学理论建模的结果（颇有些范畴论的味道），方便科学软件编写分析与开发（像极了数据库的SQL语言的描述），深感haskell是专为写编译器和软件学术研究而生的语言，尽管学习曲线陡峭，希望能将它发扬传承下去。**