Matheme函数式语言设计

小组成员

ZY2006160 王宇翔

ZY2006357 武仕沛

## 1. 问题分析

函数式程序设计语言是以数学函数为基础设计的，即从一个集合到另一个集合的映射，并且采用递归和条件表达式作为控制手段，相比于命令式语言顺序结构和循环求值显得更加自然。由于函数式程序设计语言没有模拟存储单元的变量的概念，函数副作用也因此在这里得到根除。

LISP和Scheme作为函数式程序设计语言的代表，很好的实现了上述的特性。以原子和链表作为主要的数据对象，就可以产生出强大的表达能力。Scheme在LISP的基础上进行改进，将函数作为第一类实体对待，即函数可以表示链表元素、表达式以及作为参数进行传递，使得语言特性更加丰富。

然而上述语言仍存在一些不足和缺欠。例如LISP和Scheme实质上都是无类型的语言，当程序变大时难免可读性变差、表达能力也变得模糊；LISP语言的语法风格与命令式语言相去甚远，例如在运算时使用前缀表达形式，Scheme语言在定义函数时多层嵌套等，使得熟悉命令式语言的人员在学习函数式语言时显得力不从心；在数学函数和数学集合表达方面，上述语言也表现不足。

针对上述问题，我们在LISP和Scheme语言的基础上，增加以下语言特性：

（1）加入**模式匹配**，并且将模式匹配与多种语法结构结合，产生出简洁又高效的语法表达形式；

（2）提出**链表综合**机制，构建出一套类似于数学集合标记的文法来定义集合，与数学表达更为贴切；

（3）采用**懒惰求值**方式，即一直等到需要进行求值时才会对其求值，并且基于此实现了定义无穷的数据结构；

（4）引入**强类型**机制，采用Hindley-Milner类型系统进行类型推导和检查，使得不需要进行变量声明，每个表达式的类型都能够在编译时间确定。

（5）语言还包括其他一些特性，如将函数作为第一类公民、定义高阶函数、设计了更接近于命令式语言的语法风格等。

## 2. 语法EBNF

语法推导规则参见附录A.1约定。

|  |
| --- |
| program := {exp | decls | literal}  lexeme ::= <literal> | <special>  literal ::= <integer> | <float> | <char> | <string>  special ::= ‘(’ | ‘)’| ‘,’ | ‘;’ | ‘[’ | ‘]’ | ‘\_’ | ‘{’ | ‘}’  whitespace ::= <whitechar>  whitechar ::= <newline> | <space> | <tab>  newline ::= <linefeed> | <return> | <formfeed>  return ::= ‘a carriage return’  linefeed ::= ‘a line feed’  formfeed ::= ‘a form feed’  space ::= ‘a space’  tab ::= ‘a horizontal tab’  graphic ::= <small> | <large> | <symbol> | <digit> | <specail>  small ::= <ascSmall>  ascSmall ::= ‘a’ | ‘b’ | ... | ‘z’  large ::= <ascLarge>  ascLarge ::= ‘A’ | ‘B’ | ... | ‘Z’  symbol ::= <ascSymbol> | <uniSymbol> | <special> | ‘\_’ | ‘:’ | ‘"’ | ‘>’  ascSymbol ::= ‘!’ | ‘#’ | ‘$’ | ‘%’ | ‘&’ | ‘\*’ | ‘+’ | ‘.’ | ‘/’ | ‘<’ | ‘=’ | ‘>’ | ‘?’ | ‘@’ | ‘\’ | ‘^’ | ‘|’ | ‘-‘ | ‘~’  digit ::= <ascDigit>  ascDigit ::= ‘0’ | ‘1’ | ... | ‘9’  variable ::= <small>{<small> | <large> | <digit> }  decimal ::= <digit>{<digit>}  integer ::= <decimal> | 0xdecimal  float ::= <decimal> ‘.’ <decimal> [<exponent>] | <decimal> <exponent>  exponent ::= (‘e’ | ‘E’) [‘+’ | ‘-‘] <decimal>  char ::= ' (graphic<' | \> | space | escape<\&>) '  string ::= " {graphic<" | \> | space | escape | gap}"  exp ::= <infixexp> [<type>]  infixexp ::= <lexp> <qop> <infixexp>  | <infixexp>  | <lexp>  lexp ::= ‘let’ <decls> ‘in <exp>  | ‘if’ <exp> ‘;’ ‘then’ <exp> ‘;’ ‘else’ <exp>  | ‘case’ <exp> ‘of’ <alts>  | ‘do’ <stmts>  | <fexp>  fexp ::= [<fexp>] <aexp> //Lambda Abstractions  aexp ::= <var> | <con> | <literal> | ‘(’ <exp> {<exp>} ‘)’ | ‘[’ <exp> {<exp>} ‘]’  | ‘(’ <infixexp> <qop> ‘)’  aexp ::= <var> | <con> | <literal>  var :: = <varid> | ‘(’ <varsym> ‘)’  qvar :: = <qvarid> | ‘(’ <qvarsym> ‘)’  con :: = <conid> | ‘(’ consym ‘)’  qcon :: = <qconid> | ‘(’ gconsym ‘)’  varop :: = <varsym> | ‘varid’  qvarop :: = <qvarsym> | ‘qvarid’  conop :: = <consym> | ‘conid’  qconop :: = <gconsym> | ‘qconid’  op :: = <varop> | <conop>  qop :: = <qvarop> | <qconop>  fexp ::= [<fexp>] <aexp>  alts ::= <alt> {<alt>}  alt ::= <pat> ‘->’ <exp> ‘[where’ <decls> ‘]’  | <pat> <gdpat> ‘[where’ <decls> ‘]’  gdpat ::= <guards> ‘->’ <exp> ‘[‘ <gdpat> ‘]’  stmts :: = <stmt> {<stmt>} <exp> ‘;’  stmt :: = <exp> ‘;’  | <pat> ‘<-‘ <exp> ‘;’  | ‘let’ <decls> ‘;’  | ‘;’  pat :: = <lpat> <qconop> <pat>  | <lpat>  lpat :: = <apat>  | ‘(’ <integer> | <float> ‘)’  apat :: = <var> ‘[’ <apat> ‘]’  | <gcon>  | <literal>  body :: = ‘{’ <impdecls> ‘;’ <topdecls> ‘}’  | ‘{’ <impdecls> ‘}’  | ‘{’ <topdecls> ‘}’  topdecls :: = <topdecl> {<topdec>}  topdecl :: = ‘type’ <simpletype> ‘=’ ‘type’  | ‘data’ ‘[’<context> ‘=>’ ‘]’ <simpletype> ‘[=’ <constrs> ‘]’ ‘[‘ <deriving> ‘]’  | ‘newtype’ ‘[’ <context> ‘=>’ ‘]’ <simpletype> ‘=’ <newconstr> ‘[’ <deriving> ‘]’  | decl  decls :: = ‘{’ <decl> {<decl>}  decl :: = <gendecl> | ‘(’ <funlhs> | <pat> ‘)’ <rhs>  cdecls :: = ‘{’ <cdecl> {<cdecl>}  type ::= <btype> ‘[->’ <type> ‘]’  btype ::= ‘[’ <btype> ‘]’ <atype> ‘(’ <type> <application> ‘)’  constrs ::= <const> {<constrn>}  constr ::= <con> ‘!’ <atype> {<atype>}  | ‘(’ <btype> | ‘!’ <atype> ‘)’ <conop>  | <con> {<fielddecl>}  fielddecl ::= <vars> ‘::’ ‘(’ <type> | <atype> ‘)’  gendecl ::= <vars> ‘::’ ‘[‘ <context> ‘=>’ ‘]’ <type>  vars ::= {<var>}  decl ::= ‘(’ <funlhs> | <pat> ‘)’ <rhs>  funlhs ::= <var> <apat> {<apat>}  | ‘(’ <funlhs> ‘)’ <apat> {<apat>}  rhs ::= ‘=’ <exp> ‘[’ <where> <decls> ‘]’  | <gdrhs> ‘[’ ‘where’ <decls> ‘]’  gdrhs ::= <guards> ‘=’ <exp> ‘[’ <gdrhs> ‘]’  guards ::= <guard> {<guard>}  guard ::= <pat> ‘<-’ <infixexp>  | ‘let’ <decls>  | <infixexp> |

## 3. 指称语义

### 3.1 抽象语法

|  |
| --- |
| command :: = exp  exp ::= var  | const\_var  | actual\_parameter  | constructor  | exp op exp  | tuple-expr            // tuple元组  | list-expr  | let decl in exp  | if exp then exp else exp  | pat-expr //模式匹配  | identifier infinite-expr var // 无穷数据结构  | list-cons  list-expr ::= [ {exp [,]} ]               // List链表  | [ exp1 [, exp2 ] .. [exp3 ] ]     // 算数序列  | [ exp | qual { , qual } ]    // 链表综合  list-cons := list-expr {++ list-expr}  tuple-expr ::= ( exp, exp {,exp} )  infinite-expr ::= [ exp.. ]  pat-expr := identifier exp  qual ::= exp  | pat-expr <- exp  decl ::= type var  | var = exp [where decl]  | identifier (formal\_parameter) var type exp  | pat-decl  pat-decl ::= case exp of { alts }  | identifier const\_var = exp  | identifier var = exp  | identifier tuple-expr = exp  | identifier list-expr = exp  formal\_parameter ::= type var  actual\_parameter ::= exp  type ::= Integer  | Float  | Bool  | Char  | String  | Ord  | [ type ]  | ( type1, … , typek ) k>=2 |

### 3.2 语义域说明

|  |  |
| --- | --- |
| 域名 | 含义 |
| Integer | I, 域元素为整数 |
| Float | F, 域元素为浮点数 |
| Bool | B, 布尔类型值True|False |
| Char | C, 域元素取自ASCLL字符集 |
| String | S = {C}, 由基本域C构造的复合域 |
| Ord | O, 域元素为比较规则字符 |
| Value | V = I + F + B + C + S + L + T |
| Type | A = {Integer, Float, Bool, Char, String, Ord} |
| List | L = {V}，由复合域V构造的复合域，其中元素类型均一致 |
| Tuple | T = {V}，由复合域V构造的复合域，域长度固定 |
| Id-Map | M = +I -> V，由正整数构成的基本域和复合域V构成的函数域，表示一个整数标识到变量的映射 |
| Id-E | MI = +I->E，表示二级映射，由正整数构成的基本域和环境域E组成 |
| Environ | E = {M}，环境域由id-map和id-M复合得到，表示在某一个环境中的多个映射关系。 |

说明：命令式语言指称语义中包含存储域Store=Loc->stored以及环境域Environ=Id->bindble两个特殊域，这里我们将其合并为从Id直接到value的映射关系。并且不同作用域对应不同的环境，从而在对应的环境中找到相关的标识。

### 3.3 语义函数和辅助函数说明

|  |
| --- |
| execute: command->environ->environ  elaborate: declaration->environ->environ  evaluate: expression->environ->value×environ  allocate: environ-> environ×id  deallocate: environ×id-> environ  \*辅助函数  // 在环境中找到对应的id-map映射，get\_value为底层函数  find: environ×id->value  find(env,I) =  let get\_value(map) = map  get\_value(not map) = ⊥  in get\_value(env(I))  find2: environ×id->environ // 二级映射查找  // 用一个新的环境覆盖当前环境，用来实现变量作用域和函数递归  overlay: environ×environ->environ  overlay(env’,env) =  λI. if env’(I) /= map then env’(I) else env(I)  // 表达式计算，这里用立即求值的方式，惰性求值只在定义无穷数据结构使用。  caculate: Integer×Integer->Integer, Float×Float->Float  caculate(val1, val2, op) =  if op == ‘+’ then let val = val1 + val2 in val  else if op == ‘-’ then let val = val1 – val2 in val  else if op == ‘\*’ then let val = val1 \* val2 in val  else if val2 == 0 then semantic\_error  else let val = val1 / val2 in val  // 求算数序列的辅助函数，根据起始、终结和间隔确定输出链表  get\_arith\_seq: environ->Integer×Integer×Integer->List  get\_arith\_seq(val1, gap, val2) =  if val2 == ∞ then semantic\_error  else let list = [] in  let arith\_seq\_iterator list val =  if val <= val2 then  let list = construct(val1,list) in  arith\_seq\_iterator list caculate(val,gap,+)  else  in arith\_seq\_iterator  // 声明时将标识和值绑定，第二个为二级映射绑定  bind: environ×id×value->environ  bind2: environ×id×id->value->environ  // 参数绑定辅助函数，将形参标识符与函数参数列表中的变量进行束定  bind\_argument: formal\_parameter->(argument->env)  argument = value  bind\_parameter(env,value) = bind(var, exp)  // 求实参的值，在env环境中求出变量域的值  give\_argument: actual\_parameter->(env->formal\_parameter)  give\_argument(env) =  let var I = find(env, I) in var I  // 在env环境下将多个参数值复制传值到多个参数变量上。  copy\_in: formal\_parameter | (formal\_parameter1,..) | [formal\_parameter1,..] ->(argument×env->env)  copy\_in(var val) env =  let(env’, I) = allocate env in  (bind(I, var I), env’)  // 在env环境中，将函数返回值束定到返回参数I上  copy\_out: formal\_parameter->(env×argument->env)  copy\_out(var I) env =  let var I = find(env, I) in  update(env, I, fetch(env, I))  // 更新env环境下变量I的值  update: env×I->value->env  update(env, I, value) = env[I->value]  // 读取env环境下变量I的值  fetch: env×I->value  fetch(env, I) =  let stored\_value(I) = value  stored\_value(undefined) = ⊥  stored\_value(unused) = ⊥  in  stored\_value(env(I)) |

### 3.4指称语义

列出了抽象语法中重要特性的语义说明，相似语义不再冗余列出。

|  |
| --- |
| execute {exp} env =  let {val, env’} = evaluate exp env in  env’  evaluate {var} env =  let val = find(env, var) in  (val, env)  evaluate {exp1 op exp2} env =  let (val1, env’) = evaluate exp1 env in  let (val2, env’’) = evaluate exp2 env’ in  caculate(val1, val2, op) env’’  evaluate {(exp1, exp2 {, expk})} env =  let tuple = (exp1, exp2 {,expk}) in  let evaluate\_tuple env tuple =  let (val, env’) = evaluate head(env, tuple) env in  if val /= ⊥  then let (val’, env’’) = evalulate\_tuple env’ tail(env’, tuple) in  construct(val, val’) env’’  in evaluate\_tuple  evaluate {[ {exp [,]} ] } env =  let list = [ {exp [,]} ] in  let evaluate\_list env list =  let (val, env’) = evaluate head(env, list) env in  if val /= ⊥  then let (val’, env’’) = evalulate\_list env’ tail(env’, list) in  construct(val, val’) env’’  in evaluate\_list  // 算数序列具有四种不同情况分别讨论，对于无穷序列这里在语义上不允许单独出现。  evaluate {[ exp1 [, exp2 ] .. [exp3 ] ]} env =  let (val1, env’) = evaluate exp1 env in  if null(exp2) and null(exp3) then  get\_arith\_seq(env’, val1, 1, ∞)  if null(exp3) then  let (val2, env’’) = evaluate exp2 env’ in  let (gap, \_) = caculate(val2, val1, -) in  get\_arith\_seq(env’’, val1, gap, ∞)  if null(exp2) then  let (val3, env’’) = evaluate exp3 env’ in  get\_arith\_seq(env’’, val1, 1, val3)  else let (val2, env’’) = evaluate exp2 env’ in  let (val3, env’’’) = evaluate exp3 env’’ in  let (gap, \_) = caculate(val2, val1, -) in  get\_arith\_seq(env’’’, val1, gap, val3)  // 链表综合的语义说明，将pat的操作应用于exp1上的每一个元素，然后重组元素成为一个新的链表并将其返回  evaluate {[ exp | pat<-exp1 ]  } env =  let (list, env’) = evaluate exp1 env in  let (op, env’’) = evaluate pat env’ in  let evaluate\_list env’’ list =  let (val\_head, env’’’) = evaluate head(env’’, list) env in  if val /= ⊥  then let (val’, env’’’’) = evalulate\_list env’’’ tail(env’, list) in  construct(operator(op, val\_head), val’) env’’’’  in evaluate\_list  // 定义将多个链表通过’++’的方式进行拼接的语义  evaluate {list-expr {++ list-expr}} env =  let list-cons = list-expr {++ list-expr} in  let evaluate\_cons\_list env list list2 =  let (val, env’) = evaluate list env in  if val /= ⊥  then let (val’, env’’) = evaluate\_cons\_list env’ first(list2) rest(list2) in construct(val, val’)  in evaluate\_cons\_list  // 定义无穷数据结构，使用指称语义形式化描述惰性求值，当出现无法停止的求值情况时，返回编译错误。  evaluate {identifier infinite-expr var} env =  let (val, env’) = evaluate var env in  let (val’, env’’) = evaluate infinite-expr env in  let func = find(env’’,identifier) in  // 定义递归函数，递归求值并判断当前值是否超出预定值，超出则抛出异常，避免无限求值；否则返回匹配结果，实现无穷数据结构。  let lazy\_iterator env list val =  let (val’’, env’’’) = evaluate {func head(val’) val} env’’ in  if val’’ == Bool fasle and val’ < val  then lazy\_iterotor env’’’ tail(val’) val in  else if val’’ == Bool true then val’’  else semantic\_error // 语义错误  in lazy\_iterotor  evaluate {if exp1 then exp2 else exp3} env =  let (val, env’) = evaluate exp1 env in  then evaluate exp2 env’  else evaluate exp3 env’  evaluate {let decl in exp} env =  let env’ = elaborate decl env in  evaluate exp overlay(env’, env)  // 定义函数声明的指称语义函数，在环境env下进行函数声明，得到proc与函数名proc-id的束定，参数变元与传入的值束定后，形成新的环境env’。  elaborate {declaration proc} env =  let identifier var = find(env, proc-id) in  let formal\_parameter = bind\_parameter in env’ actual\_parameter  // 模式匹配，采用二级映射的方式实现模式匹配的语义说明。首先将var和value做一层匹配，再用id和var->value做二层匹配。  elaborate {identifier const\_var = exp} env =  let (val, env’) = evaluate exp env in  let env’’ = bind(env’, const\_var, val) in  let env’’’ = bind2(env’’, identifier, const\_val) in  env’’’  // 下面三种形式的模式匹配语义说明和上述情况类似，不再冗余表示。  elaborate {identifier var = exp} env -> env’  elaborate {identifier tuple\_expr = exp} env -> env’  elaborate {identifier list\_expr = exp} env -> env’  // 模式匹配执行，二级映射查找对应值  evaluate {identifier exp} env =  let (id\_val, env’) = evaluate exp env in  let env’’ = find2(env’, identifier) in  find(env’’, id\_val)  evaluate{identifier (const) var1, identifier (const) var2...} env =  execute let var env = let(env, id1) = elaborate exp1 env in  execute let var env = let(env, id2) = elaborate exp2 env in  execute let var env = let(env, id3) = elaborate exp3 env in  ......  // 模式匹配的语义函数，alts包含任意可选匹配值，exp为待匹配变元，利用if子语义函数实现匹配成功后的执行动作，未匹配成功时，通过递归调用函数和传入剩下参数进行后续匹配，若一直未能匹配成功，则程序出错。  evaluate {case exp of { alts }} env =  if equals(exp) in  let exp = pair[0], con = pair[1] in  let pair = head(alts) then  execute con else  if null(tail(alts)) then ⊥  execute(case exp of tail(alts))  evaluate {proc exp} env =  let (exp,env) = evaluate exp env in  case of exp1 execute(exp, con1) env  case of exp2 execute(exp, con2) env  ......  evaluate {proc id(actual\_parameter)} env =  evaluate (id(actual\_parameter)) env =  let function func = find(env, id) arg in  let arg = give\_argument(actual\_parameter) env  // 赋值入语义函数，先在env环境下分配变元空间，再计算exp值，将其更新到变元上，得到新的环境。  evaluate {copy\_in(formal\_parameter exp)} env =  evaluate exp env in  let (env, value) = allocate in (bind(id, value) in  update(env, value, id)  // 赋值出语义函数，计算返回值exp，将其与返回变元进行束定，得到新环境下函数接收变元id的值。  evaluate {copy\_out(give\_argument(exp))} env =  execute copy\_out(value) env in  let var = find(env, id) in  update(env, id, fetch(env, id)) |

## 4. 语言范例及验证测试

### 4.1 模式匹配

|  |
| --- |
| // 简单的模式匹配示例  sayMe 1 = "One!"  sayMe 2 = "Two!"  sayMe 3 = "Three!"  sayMe 4 = "Four!"  sayMe 5 = "Five!"  sayMe x = "Not between 1 and 5"  // 采用模式匹配实现快速排序算法  quicksort [] = []  quicksort (x:xs) =      let smallerSorted = quicksort [a | a <- xs, a <= x]          biggerSorted = quicksort [a | a <- xs, a > x]  in  smallerSorted ++ [x] ++ biggerSorted |

### 4.2 链表综合

|  |
| --- |
| // Demo 1  [1,3..11] equal to  [1,3,5,7,9,11]  // Demo 2  [x\*2 | x <- [1..5]] equal to  [2,4,6,8,10] |

### 4.3 无穷数据结构

|  |
| --- |
| member (m:x) n      | m < n = member x n      | m == n = True      | otherwise = False  squares = [n\*n|n<-[0..]]  member squares 16 // result is True  member squares 17 // result is False |

### 4.4 类型检查

类型推导规则参见附录A.2中。

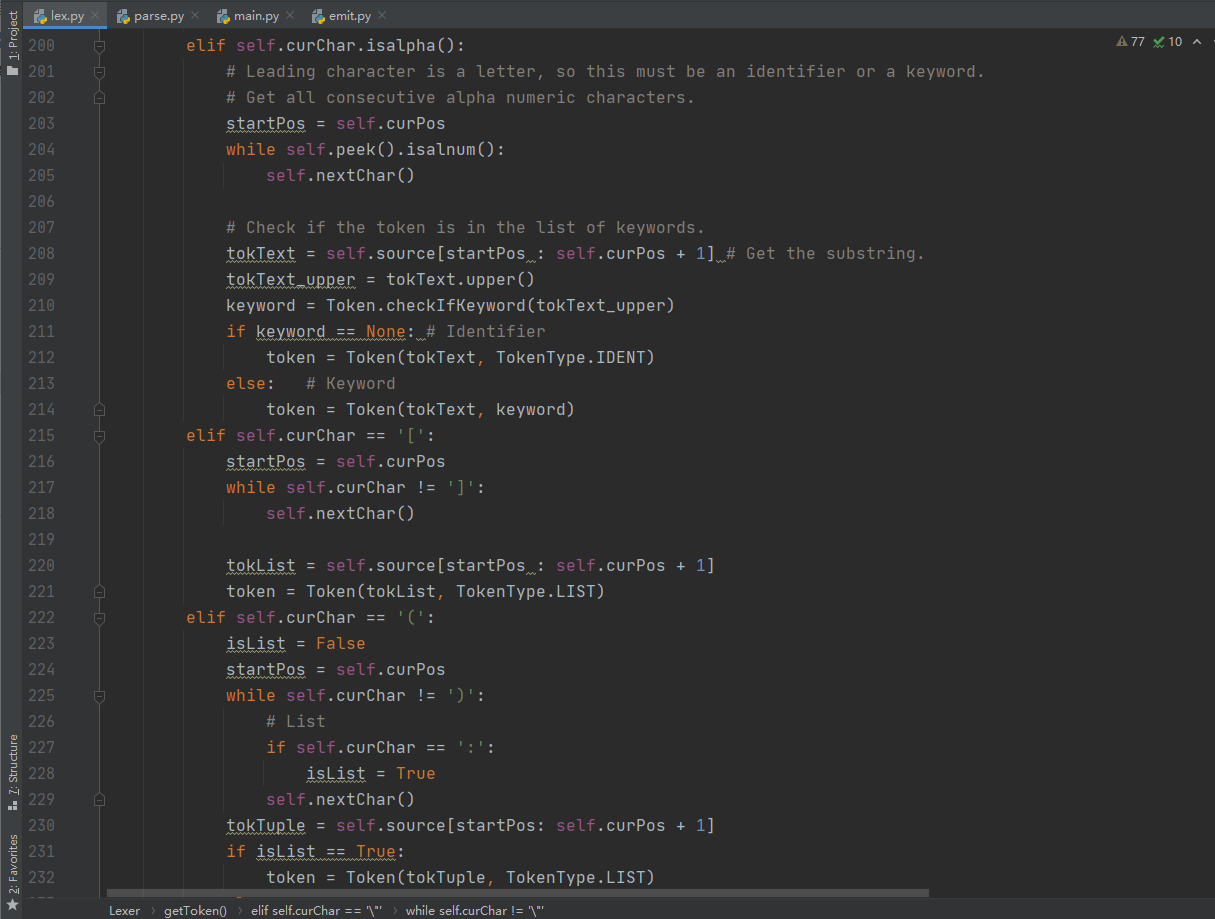
|  |
| --- |
| // 以4.1模式匹配中快速排序算法的类型推导为例  (T-Pat)  // 结果 |

### 4.5 验证测试

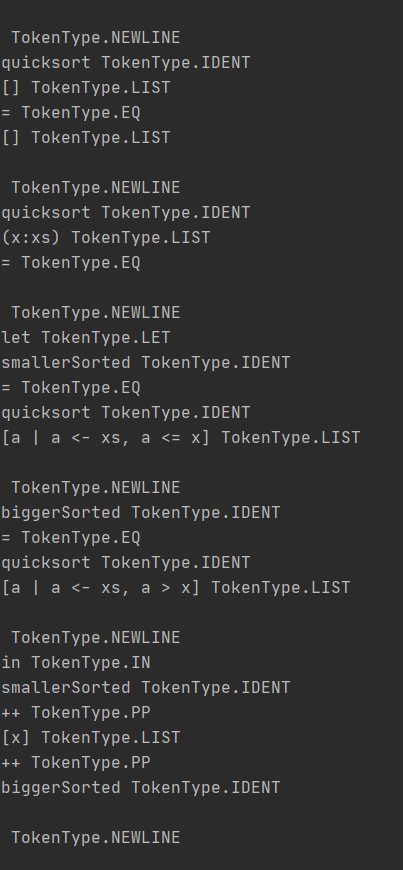
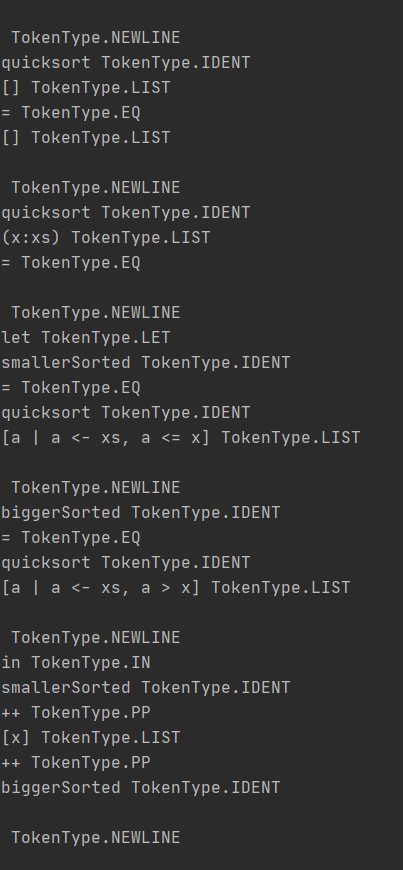
由于时间有限，我们采用Python3.8实现并验证了部分设计目标，包括Matheme的词法和语法分析，以及部分静态语义检查和类型检查。实现了基本类型包括Integer类型、Float类型、Bool类型、String类型，实现了复合类型List和Tuple，包括算数List和链表综合。实现了基于Pattern语法的模式匹配，并应用在函数定义中。

**（1）词法分析**

词法分析的目标是对输入字符进行扫描并生成具有标注的Token。定义了Lexer、Token和TokenType(Enum)三个类，主要函数包括nextChar(), peek(), abort(message), skipComment(), getToken(), checkIfKeyword(tokenText)。getToken()主要负责词法检查和生成，如下图所示部分代码。

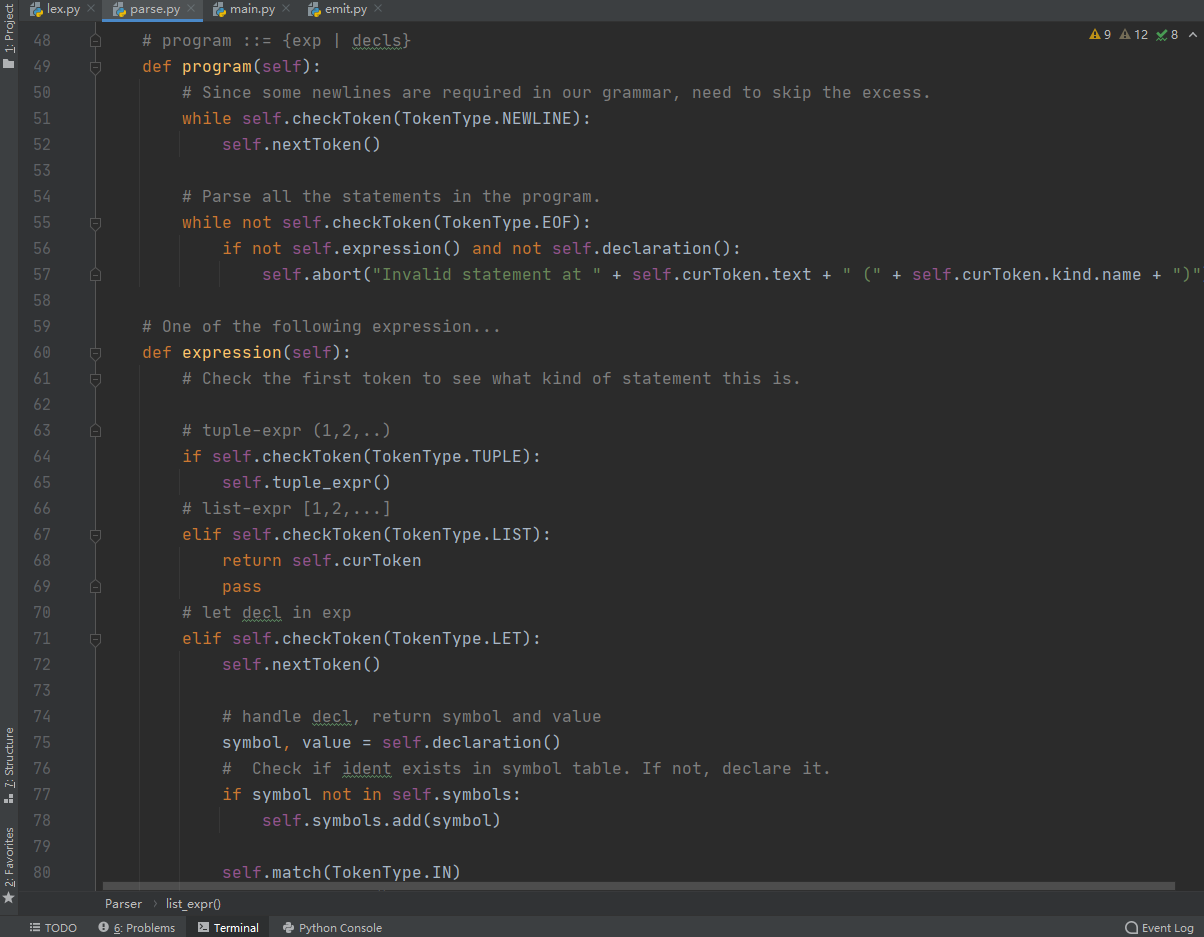


对4.1快排算法代码进行词法检查的结果如图所示。

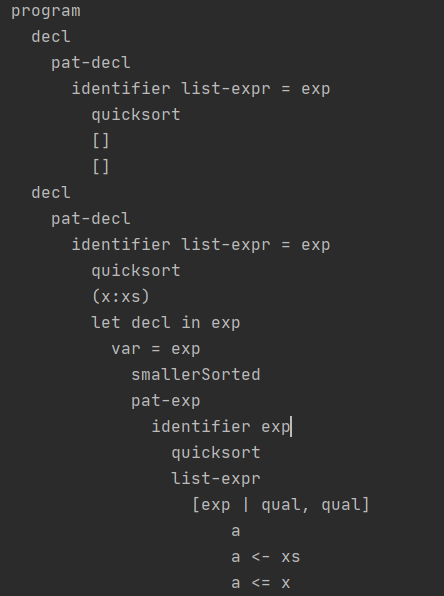
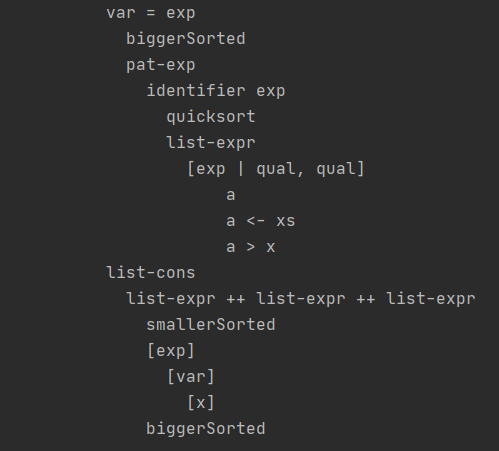
 

**（2）语法分析**

语法分析的目标是检查输入代码是否符合所定义的语法BNF，并且检查是否满足一些静态语义。以Lexer生成结果为输入，定义了Parser类，主要函数包括checkToken(kind), checkPeek(kind), match(kind), nextToken(), program(), expression(), declaration(), tuple\_expr(), list\_expr()等。如下图所示部分代码。



输出4.1快排算法程序对应的语法结构，如图所示。

## 附录A

### 1. 语法EBNF推导规则与符号表示

:= 推导

[pattern] 可选

{pattern} 0或多次

(pattern) 组

pat1 | pat2 选择

<symbol> 非终结符

‘symbol’ 终结符

### 2. 基于Hindley-Milner规则的类型系统

使用Hindley-Milner推导符号和规则系统，对Matheme进行类型推导，基本类型声明如下：

### 3. Matheme关键词表

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| case | of | do | if | then | else |
| let | in | where | proc | type | infix |
| ++ | {--} | -> | :: | ; | <- |
| , | = | => | | | -- | [ ] |