程序设计语言原理大作业

姓名	学号	分工
祁一凡	ZY1906121	语法（EBNF）
叶柏威	SY1906118	语义（指称语义）
陈凯杰	SY1906108	类型系统（类型推导）
暴明坤	SY1906416	主导设计、编译器实现

设计背景

Scheme 是动态类型语言，不能在编译时发现类型错误。

我们设计了静态类型的 Scheme：在 Scheme 的基础上，添加了 Record 和 Sumtype 类型和模式匹配，实现了基于 Hindley-Milner 类型系统的类型推导和类型检查，并可以借助类型标注支持多态递归。

语法

```
token::=  ⟨identifier⟩ | ⟨boolean⟩ | ⟨number⟩ | ⟨character⟩ | ⟨string⟩ | ' (' | ')' |
'#(' | '' ' | '`' | ',' | ',@' | '.'
quotation mark::='''
delimiter ::= ⟨whitespace⟩ | ' (' | ')' | ' ⟨quotation mark⟩ ' | ';'
whitespace ::= ⟨space⟩ |  ⟨newline⟩
space ::= ' '
newline ::= '\n'
comment ::= ';' ⟨all subsequent characters up to a line break⟩
atmosphere ::= ⟨whitespace⟩ | ⟨comment⟩
intertoken space ::= ⟨atmosphere⟩ *
identifier ::= ⟨initial⟩ ⟨subsequent⟩ * | ⟨peculiar identifier⟩
initial ::= ⟨letter⟩ | ⟨special initial⟩
letter ::= 'a' | 'b' | 'c' |...| 'z'
special initial ::= '!' | '$' | '%' | '&' | '*' | '/' | ':' | '<' | '=' | '>' |
'?' | '^' | '_' | '~'
subsequent ::= ⟨initial⟩ | ⟨digit⟩ | ⟨special subsequent⟩
digit ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
special subsequent ::= '+' | '-' | '.' | '@'
peculiar identifier ::= '+' | '-' | '...'
keyword ::= ⟨expression keyword⟩ | 'else' | '=>' | 'define' | 'unquote' |
'unquote-splicing' | ⟨type-denoter⟩ |'define-sum' |'define-record'
expression keyword ::= 'quote' | 'lambda' | 'if' | 'set! ' | 'begin' | 'cond' |
'and' | 'or' | 'case' | 'let' | 'let*' | 'letrec' | 'do' | 'delay' |
'quasiquote'| 'match'
variable ::= ⟨any ⟨identifier⟩ that isn’t also a ⟨keyword⟩ ⟩
boolean ::= '#t' | '#f'
character ::=' #\' ⟨any character⟩ | '#\' ⟨character name⟩
character name ::= 'space' | 'newline'
string ::= ⟨quotation mark⟩ ⟨string element⟩ * ⟨quotation mark⟩
```

```
string element ::= 〈any character other than 〈quotation mark〉 or '\' 〉 |
 〈quotation mark〉 |  '\ '
number ::= 〈sign〉 〈unsigned number〉
unsigned number ::= 〈unsigned integer〉 |  〈unsigned integer〉 / 〈unsigned
integer〉 | 〈decimal〉
decimal ::= 〈unsigned integer〉 | 〈digit〉 + '. ' 〈digit〉 +
unsigned integer ::= 〈digit〉 +
sign ::= 〈empty〉 | ' + ' | ' - '

datum ::= 〈simple datum〉 | 〈compound datum〉
simple datum ::= 〈boolean〉 | 〈number〉 | 〈character〉 | 〈string〉 | 〈symbol〉
symbol ::= 〈identifier〉
compound datum ::= 〈list〉 | 〈tuple〉 | 〈sumtype〉 | 〈record〉

expression ::=  〈constant〉 | 〈id〉
        | 〈left parenthesis〉 〈expression〉 〈space〉 + 〈expression〉 〈right
parenthesis〉
        |  〈lambda-expr〉
        |  〈if-expr〉
        |  〈cond-expr〉
        |  〈and-expr〉
        |  〈or-expr〉
        |  〈match-expr〉
        |  〈let-expr〉
        |  〈left parenthesis〉 'set!' 〈space〉 + 〈id〉 〈space〉 + 〈expression〉 〈right
parenthesis〉
| 〈left parenthesis〉 〈proc-id〉 ( 〈space〉 + 〈actual-param-expr〉 )* 〈right
parenthesis〉
        | 〈left parenthesis〉 〈proc-expr〉 ( 〈space〉 + 〈actual-param-
expr〉 )* 〈right parenthesis〉
lambda-expr ::= 〈left parenthesis〉 'lambda' 〈left parenthesis〉 ( 〈space〉 +
 〈formal-param-id〉 )* 〈right parenthesis〉 〈body-expr〉 〈right parenthesis〉
if-expr ::= 〈left parenthesis〉 'if ' 〈space〉 + 〈test-expr〉 〈space〉 + 〈then-expr〉
 〈space〉 + 〈else-expr〉 〈right parenthesis〉
cond-expr ::=  〈left parenthesis〉 'cond' ( 〈space〉 + 〈cond-clause〉 )* 〈right
parenthesis〉
cond-clause ::=  〈left parenthesis〉 〈test-expr〉 〈space〉 + 〈body-expr〉 〈right
parenthesis〉
        |  〈left parenthesis〉 'else' 〈space〉 + 〈body-expr〉 〈right parenthesis〉
        | 〈left parenthesis〉 〈test-expr〉 〈space〉 * '=>' 〈space〉 * 〈proc-expr〉
 〈right parenthesis〉
        | 〈left parenthesis〉 〈test-expr〉 〈right parenthesis〉
and-expr ::=  〈left parenthesis〉 'and' 〈space〉 + 〈expression〉 * 〈right
parenthesis〉
or-expr ::=  〈left parenthesis〉 'or' 〈space〉 + 〈expression〉 * 〈right
parenthesis〉
match-expr ::=   〈left parenthesis〉 'match ' 〈space〉 + 〈val-expr〉 ( 〈space〉 +
 〈match-clause〉 )* 〈right parenthesis〉
match-clause ::= 〈left parenthesis〉 〈match-pat〉 〈space〉 + 〈body-expr〉 〈right
parenthesis〉
match-pat ::=  〈id〉
        |  〈left parenthesis〉 'var' 〈space〉 + 〈id〉 〈right parenthesis〉
        |  〈left parenthesis〉 'quote' 〈space〉 + 〈datum〉 〈right parenthesis〉
        | 〈type-pat〉
        | 〈left parenthesis〉 'cons' 〈space〉 + 〈match-pat〉 〈space〉 + 〈match-pat〉
 〈right parenthesis〉
        |  〈left parenthesis〉 'and' ( 〈space〉 + 〈match-pat〉 )* 〈right
parenthesis〉
```

```
        | ⟨left parenthesis⟩  'or' ( ⟨space⟩ + ⟨match-pat⟩ )* ⟨right parenthesis⟩
        | ⟨left parenthesis⟩  'not'  ⟨space⟩ + ⟨match-pat⟩  ⟨right parenthesis⟩
        | ⟨left parenthesis⟩  '?'  ⟨space⟩ + ⟨test-expr⟩  ⟨right parenthesis⟩
        | ⟨left parenthesis⟩  '?'  ⟨space⟩ + ⟨test-expr⟩  ⟨space⟩ + ⟨match-pat⟩
 ⟨right parenthesis⟩
type-pat ::= ⟨basic-pat⟩
        | ⟨list-pat⟩
        |  ⟨tuple-pat⟩
        |  ⟨sumtype-pat⟩
        |  ⟨record-pat⟩
basic-pat ::= ⟨boolean⟩
        | ⟨number⟩
        | ⟨character⟩
        | ⟨string⟩
        | ⟨symbol⟩
        | ⟨keyword⟩
list-pat ::= ⟨left parenthesis⟩  'list'( ⟨space⟩ + ⟨match-pat⟩ )* ⟨right
parenthesis⟩
tuple-pat ::= ⟨left parenthesis⟩ 'tuple'( ⟨space⟩ + ⟨match-pat⟩ )* ⟨right
parenthesis⟩
sumtype-pat ::= ⟨left parenthesis⟩  ⟨typename-symbol⟩ ( ⟨space⟩ + ⟨match-
pat⟩ )* ⟨right parenthesis⟩
record-pat ::=  ⟨left parenthesis⟩  ⟨typename-symbol⟩ ( ⟨space⟩ + ⟨id⟩ )*  ⟨right
parenthesis⟩
define-expr ::= ⟨typed-define-expr⟩
        | ⟨untyped-define-expr⟩
| ⟨left parenthesis⟩  'define-sum' ⟨space⟩ + ⟨typename-symbol⟩ ( ⟨space⟩ + ⟨data-
symbol⟩ ( ⟨space⟩ + ⟨type-denoter⟩ )*)* ⟨right parenthesis⟩
        |  ⟨left parenthesis⟩ 'define-sum' ⟨left parenthesis⟩  ⟨typename-symbol⟩
( ⟨space⟩ + ⟨type-symbol⟩ )*  ⟨right parenthesis⟩  ( ⟨space⟩ + ⟨data-symbol⟩
( ⟨space⟩ + ⟨type-denoter⟩ )*)+  ⟨right parenthesis⟩
        |  ⟨left parenthesis⟩  'define-record' ⟨space⟩ + ⟨typename-symbol⟩  ( ⟨left
parenthesis⟩  ⟨fieldname-symbol⟩  ⟨space⟩ + ⟨type-denoter⟩  ⟨right parenthesis⟩ )
* ⟨right parenthesis⟩
typed-define-expr ::= ⟨left parenthesis⟩ 'define'  ⟨space⟩ + ⟨id⟩  ⟨space⟩ + ⟨type-
denoter⟩  ⟨space⟩ + ⟨expression⟩  ⟨right parenthesis⟩
        | ⟨left parenthesis⟩ 'define' ⟨left parenthesis⟩  ⟨id⟩ ( ⟨space⟩ + ⟨formal-
param-id⟩  ⟨space⟩ + ⟨type-denoter⟩ )*  ⟨right parenthesis⟩   ⟨type-denoter⟩
 ⟨space⟩ + ⟨expression⟩  ⟨right parenthesis⟩
untyped-define-expr ::=  ⟨left parenthesis⟩ 'define' ⟨space⟩ + ⟨id⟩  ⟨expression⟩
 ⟨right parenthesis⟩
        | ⟨left parenthesis⟩  'define'  ⟨left parenthesis⟩  ⟨id⟩ ( ⟨space⟩ +
 ⟨formal-param-id⟩ )*  ⟨right parenthesis⟩  ⟨expression⟩  ⟨right parenthesis⟩
let-expr ::=  ⟨left parenthesis⟩  'let'  ⟨left parenthesis⟩   ( ⟨left parenthesis⟩
 ⟨id⟩  ⟨space⟩ + ⟨val-expr⟩  ⟨right parenthesis⟩ ) * ⟨right parenthesis⟩   ⟨body-
expr⟩  ⟨right parenthesis⟩
        |  ⟨left parenthesis⟩  'let'  ⟨space⟩ + ⟨proc-id⟩   ⟨left parenthesis⟩
( ⟨left parenthesis⟩  ⟨param-id⟩  ⟨space⟩ + ⟨init-expr⟩  ⟨right parenthesis⟩ )*
 ⟨right parenthesis⟩    ⟨body-expr⟩  ⟨right parenthesis⟩
type-denoter ::= 'Boolean'
            | 'Number'
            | 'Char'
            | 'String'
            | 'Symbol'
            | 'List'
            | 'Tuple'
            | 'Sumtype'
            | 'Record'
```

```
             | 'Unit'
body-expr ::= ⟨define-expr⟩ ⟨expression⟩
test-expr ::= ⟨expression⟩
then-expr ::= ⟨expression⟩
else-expr ::= ⟨expression⟩
proc-expr ::= ⟨expression⟩
val-expr ::= ⟨expression⟩
init-expr ::= ⟨expression⟩
actual-param-expr ::= ⟨expression⟩
id ::= ⟨symbol⟩
proc-id ::= ⟨symbol⟩
formal-param-id ::= ⟨symbol⟩
param-id ::= ⟨symbol⟩
typename-symbol ::= ⟨symbol⟩
data-symbol ::= ⟨symbol⟩
fieldname-symbol ::= ⟨symbol⟩
left parenthesis ::=  ⟨space⟩ * ' (' ⟨space⟩ *
right parenthesis ::= ⟨space⟩ * ')' ⟨space⟩ *
constant ::=  ⟨variable⟩

program ::=  ⟨command or definition⟩ *
command or definition ::=  ⟨command⟩ |  ⟨define-expr⟩
command ::=  ⟨expression⟩
```

语义

抽象语法

```
command ::= expr
```

```
declaration ::= typed-define-statement
        | untyped-define-statement
        | (define-sum typename-symbol [data-symbol type-dennoter*]*) ; Sumtype
        | (define-sum (typename-symbol type-symbol*) [data-symbol type-
dennoter*]+) ; Sumtype
        | (define-record typename-symbol [fieldname-symbol type-dennoter]*) ;
Record
typed-define-statement ::= (define id type-dennoter expr)
        | (define (id [formal-param-id type-dennoter]*) type-dennoter expr)
untyped-define-statement ::= (define id expr)
        | (define (id formal-param-id*) expr)
```

```
expr ::= constant | id
        | if-expr
        | and-expr
        | or-expr
        | cond-expr
        | lambda-expr
        | let-expr
        | (expr expr)
        | (set! id expr)
        | (proc-id actual-param-expr*)
        | (proc-expr actual-param-expr*)
        | define-expr
        | match-expr
```

```
lambda-expr ::= (lambda (formal-param-id*) body-expr)
```

```
if-expr ::= (if test-expr then-expr else-expr)
```

```
cond-expr ::= (cond cond-clause*)
cond-clause ::= (test-expr body-expr)
        | (else body-expr)
        | (test-expr => proc-expr)
        | (test-expr)
```

```
and-expr ::= (and expr*)
```

```
or-expr ::= (or expr*)
```

```
match-expr ::= (match val-expr [match-pat body-expr]*)
match-pat ::= id
        | (var id)
        | (quote datum)
```

```
          | (cons match-pat match-pat)
          | type-pat
type-pat ::= basic-pat
          | list-pat
          | tuple-pat
          | sumtype-pat
          | record-pat
basic-pat ::= boolean-value
          | number-value
          | character-value
          | string-value
list-pat ::= (list match-pat*)
tuple-pat ::= (tuple match-pat*)
sumtype-pat ::= (typename match-pat*)
record-pat ::= (typename match-pat*)
```

```
let-expr ::= (let ([id val-expr]*) body-expr)
```

```
type-dennoter ::= Unit
          | Boolean
          | Number
          | Character
          | String
          | Symbol
          | List
          | Tuple
          | Sumtype
          | Record
```

语义域

```
unit                        U = {empty-value}
boolean                     T = {false, true}
symbol                      Q
char                        H
number                      R
string                      Es = H*
list                        El = E* ; lists中元素的类型需要一致
tuple                       Et = E* ; 长度固定
sumtype                     Em = Q × Et
record                      Er = Et
function                    F
value                       E = U + T + Q + H + R + Es + El + Et + Em + Er + F
type                        A = {unit, boolean, symbol, char, number, string,
list, tuple}
id-map                      M = id → value
environment-stack-pointer   N
environment                 V = N → M ; environment为id-map的栈，N为栈深度，整个程序
仅有一个环境栈
```

说明

sumtype 类型

```
sumtype 类型的语义域为 Em = Q × Et，例如下面这段代码：

> (define-sum Shape
>       [Rect Number Number]
>       [Circle Number])

Shape 类型变量的值将会被处理如下，实际按 tuple 来存储：

> id → [Shape.Rect (Number Number)]
> id → [Shape.Circle (Number)]
```

record 类型

```
record 类型变量在编译后会被处理成 tuple 类型，故它的语义域为 Er = Et
```

environment 环境栈

```
我们将传统语言的环境和存储
> environment: identifier → location
> store: location → value
合并为标识符映射。
> identifier-map: identifier → value

为了实现变量作用域，函数参数机制且支持函数递归，我们创建了环境栈
> environment-stack: environment-stack-pointer → identifier-map
栈中元素为标识符映射(identifier-map)，并用环境栈指针(environment-stack-pointer)来指向当前标识符映射。

当进入一个新的作用域时，会向环境栈中压入一个空的标识符映射，并更新环境栈指针。当向环境栈中寻找标识符对应的值时，会先在环境栈指针指向的标识符映射中寻找相应的标识符，若找得到则直接得到对应的值，若找不到则向栈底方向的标识符映射递归寻找，直到找到为止。
```

语义函数

```
execute: command → environment → environment
elaborate: declaration → environment → environment-stack-pointer → environment
evaluate: expression → environment → environment-stack-pointer → value × environment
```

```
execute{expr} env =
    let (val, env') = evaluate expr env 0 in
    env'
```

```
elaborate{(define id expr)} env ptr =
    let (val, env') = evaluate expr env ptr in
        bind(id, value val, env', ptr)

; (define id type-dennoter expr) 与 (define id expr) 类似

elaborate{(define (id formal-param-id*) expr)} env ptr =
  let func arg-ids =
    let (env', ptr') = create_new_env(env, ptr) in
```

```
            evaluate expr bind_parameter(formal-param-id*, arg-ids, env', ptr')
ptr'
    in
    bind(id, function func, env, ptr)

; (define (id [formal-param-id type-dennoter]*) type-dennoter expr) 与 (define (id
formal-param-id*) expr) 类似

; (define-sum typename-symbol [data-symbol type-dennoter*]*) 不在语法节点上执行

; (define-sum (typename-symbol type-symbol*) [data-symbol type-dennoter*]+) 不在语
法节点上执行

; evaluate{(define-record typename-symbol (fieldname-symbol type-dennoter)*) 不在语
法节点上执行
```

```
evaluate{constant} env ptr =
    (constant, env)
```

```
evaluate{id} env ptr =
    let val = find(env, ptr, id) in
    (val, env)
```

```
evaluate{(if test-expr then-expr else-expr)} env ptr =
    let (val, env') = evaluate test-expr env ptr in
        if val == boolean true
        then evaluate then-expr env' ptr
        else evaluate else-expr env' ptr
```

```
evaluate{(and expr*)} env ptr =
    if null(expr*) == boolean true
    then (#t, env)
    else let curr-expr = car(expr*) in
    let (val, env') = evaluate curr-expr env ptr in
        if val == boolean true
        then evaluate (and cdr(expr*)) env' ptr
        else (#f, env')
```

```
evaluate{(or expr*)} env ptr =
    if null(expr*) == boolean true
    then (#f, env)
    else let curr-expr = car(expr*) in
        let (val, env') = evaluate curr-expr env ptr in
            if val == boolean true
            then (val, env')
            else evaluate (or cdr(expr*)) env' ptr
```

```
evaluate{(cond cond-clause*)} env ptr =
    if null(cond-clause*) == boolean true
    then (empty-value, env)
    else let curr-cond-clause = car(cond-clause*) in
        let (judge, val, env') = evaluate-cond-clause env ptr in
          if judge == boolean true
```

```
        then (val, env')
        else evaluate (cond cdr(cond-clause*)) env' ptr

evaluate-cond-clause{(test-expr body-expr)} env ptr =
    let (val, env') = evaluate test-expr env ptr in
        if val == boolean true
        then cons #t (evaluate body-expr env' ptr)
        else (#f, empty-value, env')

evaluate-cond-clause{(else body-expr)} env ptr =
    cons #t (evaluate body-expr env ptr)

evaluate-cond-clause{(test-expr)} env ptr =
    let (val, env') = evaluate test-expr env ptr in
        if val == boolean true
        then (#t, val, env')
        else (#f, empty-value, env')

evaluate-cond-clause{(test-expr => proc-expr)} env ptr =
    let (val, env') = evaluate test-expr env ptr in
        if val == boolean true
        then let (func, env'') = evaluate proc-expr env' ptr in
            (#t, evaluate (func val) env'' ptr)
        else (#f, empty-value, env')
```

```
; 函数中所用的环境是定义函数时的环境，因此具有按引用传值的函数闭包特性
evaluate{(lambda (formal-param-id*) body-expr)} env ptr =
  let func arg-ids =
    let (env', ptr') = create_new_env(env, ptr) in
            evaluate body-expr bind_parameter(formal-param-id*, arg-ids, env',
ptr') ptr'
    in
    (function func, bind(id, function func, env, ptr))
```

```
evaluate{(let ([id val-expr]*) body-expr)} env ptr =
    let (env', ptr') = create_new_env(env, ptr) in
        let env'' = evaluate-bind-loop [id val-expr]* env' ptr' in
            evaluate body-expr env'' ptr'

evaluate-bind-loop{[id val-expr]*} env ptr =
    if null([id val-expr]*) == boolean true
        then env
    else let (curr-id, curr-val-expr) = car([id val-expr]*) in
        let (val, env') = evaluate curr-val-expr env ptr in
            evaluate-bind-loop cdr([id val-expr]*) bind(curr-id, value val,
env', ptr) ptr
```

```
evaluate{(expr1 expr2)} env ptr =
    let (val, env') = evaluate expr1 env ptr in
        evaluate expr2 env' ptr
```

```
evaluate{(set! id expr)} env ptr =
    let (val, env') = evaluate expr env ptr in
        (val, update(env', ptr, id, value val))
```

```
evaluate{(proc-id actual-param-expr*)} env ptr =
    let function func = find(env, ptr, proc-id) in
        let arg-values = give_arguments(actual-param-expr*, env, ptr) in
            (func arg-values, env)

evaluate{(proc-expr actual-param-expr*)} env ptr =
    let (function func, env') = evaluate proc-expr env ptr in
        let arg-values = give_arguments(actual-param-expr*, env', ptr) in
            (func arg-values, env')
```

```
evaluate{(match val-expr [match-pat body-expr]*)} env ptr =
    let (val, env') = evaluate val-expr env ptr
    evaluate-match-loop (match val [match-pat body-expr]*) env' ptr

evaluate-match-loop{(match val [match-pat body-expr]*)} env ptr =
    if null([match-pat body-expr]*) == boolean true
    then (empty-value, env)
    else let (curr-match-pat, curr-body-expr) = car([match-pat body-expr]*) in
        let (env', ptr') = create_new_env(env, ptr) in
            let (judge, env'') = evaluate-pattern-match val curr-match-pat env'
ptr' in
                if judge == boolean true
                then evaluate curr-body-expr env'' ptr'
                else evaluate-match-loop (match val cdr([match-pat body-expr]*))
env ptr

evaluate-pattern-match{(val id)} env ptr =
    (#t, bind(id, val, env, ptr))

evaluate-pattern-match{(val (var id))} env ptr =
    (#t, bind(id, val, env, ptr))

evaluate-pattern-match{(val (cons match-pat1 match-pat2))} env ptr =
    let list-pat = cons(match-pat1, match-pat2) in
        evaluate-pattern-match val list-pat env ptr

evaluate-pattern-match{(val (and match-pat*))} env ptr =



evaluate-pattern-match{(val boolean-value)} env ptr =
    ((evaluate-pattern-match-compare-type val boolean-value env ptr), env)

; evaluate-pattern-match{(val number-value)} 与 evaluate-pattern-match{(val
boolean-value)} 类似

; evaluate-pattern-match{(val character-value)} 与 evaluate-pattern-match{(val
number-value)} 类似

; evaluate-pattern-match{(val string-value)} 与 evaluate-pattern-match{(val
number-value)} 类似

evaluate-pattern-match{(val (list match-pat*))} env ptr =
    if (evaluate-pattern-match-compare-type val number-value env ptr) == boolean
true
    then (#t, (evaluate-pattern-match-bind-symbol val number-value env ptr))
    else (#f, env)
```

```
; evaluate-pattern-match{(val (tuple match-pat*))} 与 evaluate-pattern-match{(val
(list match-pat*))} 类似

; evaluate-pattern-match{(sumtype-type-label val (typename match-pat*))} 与
evaluate-pattern-match{(val (list match-pat*))} 类似

; evaluate-pattern-match{(record-type-label val (typename match-pat*))} 与
evaluate-pattern-match{(val (list match-pat*))} 类似

evaluate-pattern-match-compare-type{(val pat)} env ptr =
    let val-type = value_type(val) in
    let pat-type = value_type(pat) in
        if pat-type == val-type
        then if val-type == type list
            then if null(val) == boolean true
                then if null(pat) == boolean true
                    then #t
                    else #f
                else if null(pat) == boolean true
                    then #f
                    else if evaluate-pattern-match-compare-type car(val)
car(pat) env ptr == true
                        then evaluate-pattern-match-compare-type cdr(val)
cdr(pat) env ptr
                        else #f
            else if val-type == type tuple
                then if null(val) == boolean true
                    then if null(pat) == boolean true
                        then #t
                        else #f
                    else if null(pat) == boolean true
                        then #f
                        else if evaluate-pattern-match-compare-type car(val)
car(pat) env ptr == true
                            then evaluate-pattern-match-compare-type cdr(val)
cdr(pat) env ptr
                            else #f
                else if val == pat ; 当两者类型相同，且不为 list 或 tuple，那么 pat 为
常量的基础类型
                    then #t
                    else #f
        else if pat-type == type symbol ; 当两者类型不同，但 pat 为 symbol 时，视作匹
配
            then #t
            else #f

evaluate-pattern-match-bind-symbol{(val pat)} env ptr =
; 将 pat 和 val 进行束定。注意 sumtype 与 record 均被处理成 tuple
    let val-type = value_type(val) in
        let pat-type = value_type(pat) in
            if pat-type == val-type
            then if null(val) == boolean true
                then env
                else let env' = evaluate-pattern-match-bind-symbol car(val)
car(pat) env ptr in
                    evaluate-pattern-match-bind-symbol cdr(val) cdr(pat) env'
ptr
```

```
        else if pat-type == type symbol ; 当两者类型不同，pat 可能为 symbol 或简单类型常量
            then bind(pat, val, env, ptr) ; 若为 symbol，直接将 pat 束定到 val
        else env ; 简单类型常量不处理
```

辅助函数

```
; 判断表达式 list 或 tuple 是否为空
null: list → boolean
     | tuple → boolean
```

```
; 合并 value 和 list
cons: value × list → list
```

```
; 取 list 或 tuple 第一个元素
car: list → value
     | tuple → value
```

```
; 取 list 或 tuple 除第一个元素外的其他元素
cdr: list → list
     | tuple → tuple
```

```
; 从环境栈中按 id 取值，若当前 ptr 栈层找不到该 id 束定，则向下一层环境继续递归查找
find: environment × environment-stack-pointer × id → value
```

```
; 从环境栈中查找 id 束定，若当前 ptr 栈层找不到，则向下一层继续递归查找，找到后用 value 更新束定值。
update: environment × environment-stack-pointer × id × value → environment
```

```
; 向环境栈中压入一个新的 empty-id-map，并更新 ptr
create_new_env: environment × environment-stack-pointer → environment × environment-stack-pointer
```

```
; 将 id 束定于环境栈 ptr 层 id-map 的 value 上
bind: id × value × environment × environment-stack-pointer → environment
```

```
; 对参数进行束定
bind_parameter: formal-param-ids × arg-ids × environment × environment-stack-pointer → environment
```

```
; 在环境中对实参表达式 actual-param-exprs 求值
give_arguments: actual-param-exprs × environment × environment-stack-pointer → values
```

```
; 获取 value 的类型
value_type: val → type
```

类型系统

类型推导

概述

使用 Hindley-Milner 类型系统，实现类型推导和类型检查。

记法

规则公式

$$\frac{premise_1 \quad \ldots \quad premise_n}{conclusion}\text{NAME}$$

符号表

符号	含义
e	表达式
x	变量
τ	单态类型（`monotype`）
σ	多态类型（`polytype`）
Γ	环境
$\Gamma \vdash e : \tau$	在 Γ 中 e 的类型是 τ
$\Gamma, x : \sigma$	在 Γ 中加入 $x : \sigma$ 后构成的新环境
$\Gamma \vdash^{gen} \sigma \succeq \tau$	在 Γ 中将 τ 泛化成 σ
$\Gamma \vdash^{inst} \sigma \succeq \tau$	在 Γ 中将 σ 实例化成 τ

类型泛化和实例化

多态类型 σ 可以定义为：

$$\begin{array}{rcl} \sigma & = & \tau \\ & | & \forall \overline{\alpha}. \tau \end{array}$$

其中 $\overline{\alpha}$ 代表其所有参数的集合。

类型泛化：将所有不在 Γ 的自由类型变量集合中的 τ 的自由类型变量作为 τ 的参数。

$$\mathbf{gen}(\Gamma, \tau) = \begin{cases} \forall \overline{\alpha}. \tau & \mathbf{free}(\tau) - \mathbf{free}(\Gamma) = \overline{\alpha} \\ \tau & \mathbf{free}(\tau) - \mathbf{free}(\Gamma) = \Phi \end{cases}$$

在 Γ 中将 τ 泛化成 σ，记作：

$$\Gamma \vdash^{gen} \sigma \succeq \tau$$

类型实例化：用不属于 Γ 的全新类型变量替换 σ 的参数。

$$\mathbf{inst}(\Gamma, \sigma) = \begin{cases} [\overline{\alpha} \mapsto \overline{\tau}]\, \tau & \forall \overline{\alpha}. \tau \\ \tau & \tau \end{cases}$$

在 Γ 中将 σ 实例化成 τ，记作：

$$\Gamma \vdash^{inst} \sigma \succeq \tau$$

规则

variable lookup

$$\frac{x : \sigma \in \Gamma \quad \Gamma \vdash^{inst} \sigma \succeq \tau}{\Gamma \vdash x : \tau}\text{VAR}$$

含义：在 Γ 中找到变量 x 后，其类型是 σ，求值时将 σ 实例化成 τ。

abstraction

```
(lambda (x-1 ... x-n) e)
```

$$\frac{\Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e : \tau_r}{\Gamma \vdash (\textbf{lambda}\ (x_1 \ldots x_n)\ e) : \tau_1, \ldots, \tau_n \rightarrow \tau_r}\text{ABS}$$

含义：函数的类型是“参数类型 $\rightarrow$ 结果类型”。函数类型、参数类型、结果类型均为单态类型。

application

```
(e e-1 ... e-n)
```

$$\frac{\Gamma \vdash e : \tau_1 \ldots \tau_n \rightarrow \tau, e_1 : \tau_1, \ldots, e_n : \tau_n}{\Gamma \vdash (e\ e_1\ \ldots\ e_n) : \tau}\text{APP}$$

含义：应用函数时，参数和返回值的类型与函数类型对应，且均为单态类型。

let

```
(let ((x-1 e-1)
       ...
      (x-n e-n))
    e)
```

$$\frac{\Gamma \vdash e_1 : \tau_1, \ldots, e_n : \tau_n \quad \Gamma \vdash^{gen} \sigma_1 \succeq \tau_1, \ldots, \sigma_n \succeq \tau_n \quad \Gamma, x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash e : \tau}{\Gamma \vdash (let\ ((x_1\ e_1) \ldots (x_n\ e_n))\ e) : \tau}\text{LET}$$

含义：let 绑定时，将 e_i 的类型 τ_i 泛化成 σ_i，将所有 $x_i : \sigma_i$ 添加到 Γ 中，然后对 e 进行求值。

仅当一个函数绑定到一个变量时，其的类型被泛化。

define

```
(define x e)
```

$$\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash^{gen} \sigma \succeq \tau}{\Gamma' = \Gamma, x : \sigma}\text{DEF}$$

含义：define 时，将 e 的类型 τ 泛化成 σ，将 $x : \sigma$ 添加到 Γ 中生成新的环境 Γ'。

define function 是 define symbol 的语法糖

```
(define (x x-1 ... x-n) e)
```

上式等价于：

```
(define x (lambda (x-1 ... x-n) e))
```

define 可选类型标注，带有类型标注的 define 支持多态递归。

```
(define x t-x e)
```

```
(define (x [x-1 t-1] ... [x-n t-n]) t-r e)
```

set!

```
(set! x e)
```

$$\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash^{gen} \sigma \succeq \tau}{x : \sigma \in \Gamma}\text{SET}$$

含义：set! 时，e 的类型 τ 泛化成 σ，$x : \sigma$ 应当已经存在于当前环境 Γ 中。改变变量的值，变量类型不变。

match

```
(match e
       (p-1 e-1)
       ...
       (p-n e-n))
```

$$\frac{\dfrac{\Gamma \vdash p_1 : \tau', \ldots, p_n : \tau'}{\Gamma \vdash e : \tau'} \quad x_{ij} : \tau_{ij} \in p_i \quad \Gamma, \overline{x_i : \tau_i} \vdash e_i : \tau}{\Gamma \vdash (\textbf{match}\, e\ (p_1\ e_1)\ \ldots\ (p_n\ e_n)) : \tau}\text{MATCH}$$

含义：match 匹配模式 p_i 后，p_i 中的所有变量 $x_{ij} : \tau_{ij}$ 的集合写作 $\overline{x_i : \tau_i}$。所有模式 p_i 的类型应与 e 的类型相同，且是单态类型 τ'。各个 $\overline{x_i : \tau_i}$ 加入 Γ 后的环境中求值 e_i 的类型与 match 式子的类型 τ 相同。

推导过程

见示例程序中的 类型推导过程 小节。

类型

类型	含义
Unit	空值
Symbol	符号
Bool	布尔类型 `#t` 或 `#f`
Number	数值
Char	字符
String	字符串

类型	含义
List	复合类型：列表，不定长度，元素类型相同
Tuple	复合类型：元组，固定长度，元素类型可以不同
Sumtype	复合类型：多种类型的并集
Record	复合类型：结构体

预定义符号的类型

符号	类型
`if`	$\mathbf{Bool}, \tau, \tau \to \tau$
`begin`	$\tau_1, \ldots, \tau_n, \tau \to \tau$
`cons`	$\tau, \mathbf{List}[\tau] \to \mathbf{List}[\tau]$
`car`	$\mathbf{List}[\tau] \to \tau$
`cdr`	$\mathbf{List}[\tau] \to \mathbf{List}[\tau]$
`+`, `-`, `*`, `/`	$\mathbf{Number}, \mathbf{Number} \to \mathbf{Number}$
`and`, `or`	$\mathbf{Bool}^* \to \mathbf{Bool}$
`not`	$\mathbf{Bool} \to \mathbf{Bool}$
`>`, `<`, `=`, `>=`, `<=`	$\mathbf{Number}, \mathbf{Number} \to \mathbf{Bool}$
`rand`	$\mathbf{Unit} \to \mathbf{Number}$
`print`	$\tau \to \mathbf{Unit}$
`eq?`	$\tau, \tau \to \mathbf{Bool}$
`null`	$\mathbf{List}[\tau]$

说明

`cond` 是 `if` 的语法糖。

```
(cond (cond-1 expr-1)
      ...
      (cond-n expr-n)
      (else expr-else))
```

上式等价于

```
(if cond-1 expr-1
           (if ... ...
                   (if cond-n expr-n
                              expr-else)...))
```

示例程序

List 高阶函数

- `foldr`
- `concat`
- `flatten`

```
(define (foldr f x0 l)
  (match l
     [(Cons x xs) (f x (foldr f x0 xs))]
     [(Nil) x0]))

(define (concat x y) (foldr cons y x))

(define (flatten x)
  (foldr (lambda (l r) (concat l r))
     null x))

(define nest '((1 2 3) (2 3) (1)))

(print (flatten nest))
```

类型推导过程

以 `foldr` 的类型推导过程为例，书写时省略了应用 VAR 规则的步骤。

$$\frac{\Gamma \vdash \text{foldr} : \tau_1, \tau_2, \tau_3 \rightarrow \tau_4, \text{f} : \tau_1, \text{x0} : \tau_2, \text{xs} : \tau_3}{(\text{foldr f x0 xs}) : \tau_4}\text{APP}$$

$$\frac{\dfrac{\dfrac{\Gamma \vdash (\text{Cons x xs}) : \mathbf{List}[\tau_5], \text{Nil} : \mathbf{List}[\tau_5]}{\Gamma \vdash l : \mathbf{List}[\tau_5]} \quad \dfrac{\Gamma \vdash \text{f} : \tau_5, \tau_6 \rightarrow \tau_4, \text{x} : \tau_5, (\text{foldr f x0 xs}) : \tau_6}{\Gamma \vdash (\text{f x (foldr f x0 xs)}) : \tau_4}\text{APP} \quad \Gamma \vdash \text{x0} : \tau_4}{\Gamma, \text{f} : \tau_1, \text{x0} : \tau_2, \text{l} : \tau_3 \vdash (\text{match ...}) : \tau_4}\text{MATCH}}{\Gamma \vdash (\text{lambda (f x0 l) (match ...)}) : \tau_1, \tau_2, \tau_3 \rightarrow \tau_4}\text{ABS}$$

1. $(\text{foldr f x0 xs}) : \quad \tau_6 = \tau_4$
2. $\text{f} : \quad \tau_1 = \tau_5, \tau_4 \rightarrow \tau_4$
3. $\text{l} : \quad \tau_3 = \mathbf{List}[\tau_5]$
4. $\text{x0} : \quad \tau_2 = \tau_4$

$$\frac{\Gamma \vdash (\text{lambda (f x0 l) (match ...)}) : \tau \quad \Gamma \vdash^{gen} \sigma \succeq \tau}{\Gamma' = \Gamma, \text{foldr} : \sigma}\text{DEF}$$

5. $\tau = (\tau_5, \tau_4 \rightarrow \tau_4), \tau_4, \mathbf{List}[\tau_5] \rightarrow \tau_4$
6. $\sigma = \forall\alpha\forall\beta.\,(\alpha, \beta \rightarrow \beta), \beta, \mathbf{List}[\alpha] \rightarrow \beta$

类型推导结果

```
defined type List :: List a
define: foldr :: forall n.o => (o -> n -> n) -> n -> List o -> n
define: concat :: forall w => List w -> List w -> List w
define: flatten :: forall bh => List (List bh) -> List bh
define: nest :: List (List Number)
expr: (print (flatten nest)) :: Unit
```

树的遍历和树上map

- `pre-order`, `in-order`, `post-order`
- `tree-map`

```
(define-sum (Tree a)
  [Branch a (Tree a) (Tree a)]
  [Leaf a])

(define (concat x y)
  (match x
    [(Cons x xs) (cons x (concat xs y))]
    [(Nil) y]))

(define (pre-order t)
  (match t
    [(Tree.Branch v left right)
     (cons v (concat (pre-order left) (pre-order right)))]
    [(Tree.Leaf x) (cons x null)]))

(define (in-order t)
  (match t
    [(Tree.Branch v left right)
     (concat (in-order left) (cons v (in-order right)))]
    [(Tree.Leaf x) (cons x null)]))

(define (post-order t)
  (match t
    [(Tree.Branch v left right)
     (concat (post-order left) (concat (post-order right) (cons v null)))]
    [(Tree.Leaf x) (cons x null)]))

(define (tree-map f t)
  (match t
    [(Tree.Branch v left right)
     (Tree.Branch (f v) (tree-map f left) (tree-map f right))]
    [(Tree.Leaf v) (Tree.Leaf (f v))]))
```

类型推导结果

```
defined type List :: List a
defined type Tree :: Tree a
get func Tree.Branch :: a -> Tree a -> Tree a -> Tree a
get func Tree.Leaf :: a -> Tree a
define: concat :: forall q => List q -> List q -> List q
define: pre-order :: forall bk => Tree bk -> List bk
define: in-order :: forall ce => Tree ce -> List ce
define: post-order :: forall db => Tree db -> List db
define: tree-map :: forall dv.dw => (dv -> dw) -> Tree dv -> Tree dw
```

表达式树

```
(define-sum Term
  [Add Term Term]
  [Sub Term Term]
  [Mul Term Term]
  [Div Term Term]
  [Val Number])

(define (eval-term term)
  (match term
     [(Term.Add x y) (add-term x y)]
     [(Term.Sub x y) (sub-term x y)]
     [(Term.Mul x y) (mul-term x y)]
     [(Term.Div x y) (div-term x y)]
     [(Term.Val x) x]))

(define (term-combiner f)
  (lambda (x y)
    (f (eval-term x)
       (eval-term y))))

(define add-term (term-combiner +))
(define sub-term (term-combiner -))
(define mul-term (term-combiner *))
(define div-term (term-combiner /))

(define tree-1
  (Term.Div (Term.Add (Term.Val 6)
                (Term.Mul (Term.Val 2)
                  (Term.Val 3)))
        (Term.Sub (Term.Val 5)
                (Term.Val 1))))

(print (eval-term tree-1))
```

类型推导结果

```
defined type List :: List a
defined type Term :: Term
get func Term.Add :: Term -> Term -> Term
get func Term.Sub :: Term -> Term -> Term
get func Term.Mul :: Term -> Term -> Term
get func Term.Div :: Term -> Term -> Term
get func Term.Val :: Number -> Term
comps: [{'sub-term', 'add-term', 'div-term', 'mul-term', 'eval-term', 'term-
combiner'}, {'tree-1'}]
define: tree-1 :: Term
define: sub-term :: Term -> Term -> Number
define: add-term :: Term -> Term -> Number
define: div-term :: Term -> Term -> Number
define: mul-term :: Term -> Term -> Number
define: eval-term :: Term -> Number
define: term-combiner :: (Number -> Number -> Number) -> Term -> Term -> Number
expr: (print (eval-term tree-1)) :: Unit
```

类型检查和编译器的实现

类型检查器和编译器用 Python 实现，共计 3500+ 行，项目地址：https://github.com/bravomikekilo/Tscheme

项目实现了基本的Hindley-Milner类型系统和类型检查功能 实现的基本类型有 Unit类型(C语言中的void), Number类型, Bool类型， String类型， Char类型，Symbol类型

实现的其他类型有

- 函数类型
- 和类型
- 元组类型
- 记录类型(record)

在和类型的基础上, 我们对 scheme 中的 list 进行了建模, 提供了 List 类型

项目实现的语法有: if, cond, apply, let, define, set!, begin, list, tuple, match

基于我们自己设计的基于类型的match语法,我们实现现代语言中的 一个重要特性：**模式匹配**

一个常见的 patten match

```
(define (foldr f x0 l) (match l
    [(Cons x xs) (f x (foldr f x0 xs))]
    [(Nil) x0]))
```

架构上采用了两层分层中间表达形式的设计方案，这样有利于在开发中改变语法。

在解析器的实现上，采用了Parser combinator的方案，好处是可以避免引入其它依赖，同时可以直接解析生成语法对象，避免了额外的转换。