最简 $\mathscr{S}$ 语言的 Haskell 实现与应用

叶志浩

09016319

2019 年 1 月

摘要

Computability, Complexity, and Languages 一书中定义了一种 $\mathscr{S}$ 语言。该语言只有 4 条基本指令，但作者利用这些指令写出一些基本的程序，并将这些基本指令组织成宏，定义出函数，实现原始递归函数及初始函数，进而写出千变万化的程序。本报告将效仿这个过程，利用著名的函数式编程语言 Haskell 实现书中所提的每一个概念，并将其用于实际的一些计算。我们将看到，拥有 Monad 的 FP 语言，是如何恰如其分地将各类数学概念进行表现的。

目录

Part I

Introduction

本章将对 *Computability, Complexity, and Languages* 一书中的各类概念进行描述，这些概念将在下面几节中通过 Haskell 语言精确定义。

1 Programs and Computable Functions

1.1 A Programming Language

$\mathscr{S}$ 语言由以下几个要素组成：

- 变量：由 X, Y, Z 表示，X_i 代表输入，Y_i 代表输出，Z_i 代表临时变量。
- 标号：由 A, B, C 表示，其中 E 代表退出标号。
- 指令：$\mathscr{S}$ 只支持以下四种基本指令：

指令	释义
$V \leftarrow V + 1$	自增变量 V
$V \leftarrow V - 1$	自减变量 V，若 V 为 0 则不变
$V \leftarrow V$	空指令
$IF\ V \neq 0\ GOTO\ L$	若变量值不为 0，则跳转至 L，否则为下条指令

1.2 Macro

可以将若干条指令与标号组织在一起，形成的程序段被称作一个宏。例如

$$Z \leftarrow Z + 1$$
$$IF\ Z \neq 0\ GOTO\ L$$

定义了一个一个相当于 $GOTO\ L$ 指令的宏。

当宏被使用时，其中的内容被展开，其内变量、标号名需要重写，以防发生冲突。

1.3 Syntax and Computation

下面定义 $\mathscr{S}$ 语言的语法与计算规则。

语法 –

变量或**标号**的定义与前面相同。

语句是四条基本指令的其中之一，或是一个宏。

指令是一条语句，或是前附了标号 $[L]$ 的语句。

程序$\mathscr{P}$ 是指令的有序列表。

程序的长度 n 是指令列表的长度。

程序的状态 σ 是各变量的赋值集合。

程序的快照 s 是 (指令序号，程序状态) 的键值对。(n, σ) 代表程序的**终止**。

计算 –

一个快照 (i, σ) 的**后继** (j, τ) 被定义为 $\mathscr{P}$ 的第 i 条指令按规则执行后，程序的当前指令序号 j 与状态 τ（执行规则参见对应原文）。

一个程序 $\mathscr{P}$ 的**计算**被定义为一个快照的序列 $s_1, s_2, ..., s_k$，其中 s_{i+1} 是 s_i 的后继，s_k 则是程序的终止。

1.4 Computable Functions

函数的参数 我们通过设置程序的**起始状态** σ：$\{X_1 = r_1, X_2 = r_2, ..., X_m = r_m, Y = 0\}$，从而向程序传入了 m 个函数参数 $r_1, r_2, ...r_m$。

设函数接受 m 个参数，最终传入了 n 个参数，则当 $m < n$ 时，剩下的未传入的参数将被设为 0。当 $m > n$ 时，多余的参数将被忽略。

函数调用宏 我们将 $\mathscr{P}$ 写为

$$\mathscr{P} = \mathscr{P}(Y, X_1, ...X_n, Z_1, ...Z_k; E, A_1, ..., A_l)$$

则我们可以通过重命名 $\mathscr{P}$ 中的所有符号，得到一个新的程序：

$$\mathscr{Q}_m = \mathscr{P}(Z_m, Z_{m+1}, ...Z_{m+n}, Z_{m+n+1}, ...Z_{m+n+k}; E_m, A_{m+1}, ..., A_{m+l})$$

其中，m 是使得符号不重复的任意大的整数。

基于上面的符号重写机制，我们便可以着手编写函数调用的宏指令：

$$W \leftarrow f(V_1, ...V_n)$$

按以下方式展开即可：

$$\begin{array}{ll} & Z_m \leftarrow 0 \ ; \ Output \\ & Z_{m+1} \leftarrow V_1 \ ; \ Inputs \\ & ... \\ & Z_{m+n} \leftarrow V_n \\ & Z_{m+n+1} \leftarrow 0 \ ; \ Locals \\ & ... \\ & Z_{m+n+k} \leftarrow 0 \\ & \mathscr{Q}_m \\ [E_m] & W \leftarrow Z_m \end{array}$$

2 Primitive Recursive Functions

2.1 Composition and Recursion

组合 1-1 阶的组合即常见的复合函数：

$$h(x) = f(g(x)) = (f \circ g)(x)$$

$k - n$ 阶的组合定义如下：给定 1 个 k 元函数 f 及 k 个 n 元函数 $g_1, ..., g_k$，则

$$h(x_1, ..., x_n) = f(g_1(x_1, ..., x_n), ..., g_k(x_1, ..., x_n))$$

称为 f 与 $g_1, ..., g_k$ 的**组合**。

这个函数可以通过如下的程序计算得到：

$$\begin{aligned}
&Z_1 \leftarrow g_1(X_1, ..., X_n) \\
&... \\
&Z_k \leftarrow g_k(X_1, ..., X_n) \\
&Y \leftarrow f(Z_1, ...Z_k)
\end{aligned}$$

递归　设 $f() = k$ 是一个常函数，g 是一个 2 元函数，则

$$\begin{aligned}
h(0) &= k \\
h(t+1) &= g(t, h(t))
\end{aligned}$$

称为 h 是由 g 的**原始递归**得到的，或简称**递归**。

更一般情况的原始递归如下所示：

$$\begin{aligned}
h(x_1, ...x_n, 0) &= f(x_1, ...x_n) \\
h(x_1, ...x_n, t+1) &= g(t, h(x_1, ..., x_n, t), x_1, ..., x_n)
\end{aligned}$$

其中：

- h 称为由原始递归得到的 $n+1$ 元函数。
- f 是一个 n 元函数，称作 Base Case。
- g 是一个 $n+2$ 元函数，称作 Recursive Step。

这个函数可以通过如下的程序计算得到：

$$\begin{aligned}
& Y \leftarrow f(X_1, ..., X_n) \\
[A]\quad & IF\ X_{n+1} = 0\ GOTO\ E \\
& Y \leftarrow g(Z, Y, X_1, ..., X_n) \\
& Z \leftarrow Z + 1 \\
& X_{n+1} \leftarrow X_{n+1} - 1 \\
& GOTO\ A
\end{aligned}$$

2.2 Initial Functions and PRC Class

定义 1 以下三个函数被称为**初始函数**：

零函数 $z() = 0$，是一个 *0* 元函数（即常函数）。

后继函数 $s(x) = x + 1$，是一个 *1* 元函数。

投影函数 $u_i^n(x_1, ..., x_n) = x_i,\ i \in [0, n)$，是一个 n 元函数。

定义 2 当一个全函数的类 $\mathscr{C}$ 满足以下条件时：

1. 初始函数属于 $\mathscr{C}$
2. 由 $\mathscr{C}$ 中的函数经组合、递归后得到的函数属于 $\mathscr{C}$

则称为 $\mathscr{C}$ 为一个**原始递归封闭类**（*PRC Class*）

关于 PRC Class，可以证明以下定理以及推论：

定理 2.1 由可计算的函数构成的类是 *PRC Class*。

推论 1 由原始递归函数构成的类是 *PRC Class*。

这就是说，我们可以通过 $\mathscr{S}$ 语言实现任意一个原始递归函数。

2.3 Pairing Function and Godel Numbers

Pairing Function 一个整数对 $< x, y >$ 的编码定义如下：

$$< x, y > = 2^x(2y + 1) \dot{-} 1$$

Tuple Coding 一个序列的**哥德尔数**（即序列的编码）定义如下：

$$[a_1, ..., a_n] = \prod_{i=1}^{n} p_i^{a_i}$$

Part II

Implementation

本章将利用 Haskell 语言对章节 1 中所涉及到的各种概念进行定义与实现，组织成一个功能模块。

3 Program

3.1 Type Definition

我们逐一定义 $\mathscr{S}$ 语言中涉及到的各类概念：

值与地址 值可以是全体自然数，故为 Integer; 地址为 32 位整数，故为 Int。

```haskell
type Value = Integer
type Address = Int
```

变量与标号 整数用作变量与标号的索引，但它们并不是纯整数，故用 newtype 声明。

```haskell
newtype Variable = Var Int deriving (Show, Eq, Ord)
newtype Label = Label Int deriving (Show, Eq, Ord)
```

指令与程序 尽管 $\mathscr{S}$ 语言中只使用了 4 条基本指令，但仅使用这些指令构造出来的程序性能实在羸弱，因此增设了一些辅助指令。

```haskell
data Instruction
    = Nop
    | Inc Variable
    | Dec Variable
    | Gnz Variable Label
    | Set Variable Value
    | Mov Variable Variable
    deriving (Show)
type Program = [Instruction]
```

状态与快照　本实现使用 Data.Map 作为变量与标号的符号表，以描述键值对映射。

```haskell
data State = State {
    varTable :: Map Variable Value,
    labelTable :: Map Label Address
} deriving (Show)
type Snapshot = (Int, State)
```

程序运行环境　本实现使用 (程序, 状态, 退出标号) 的三元组唯一标记当前的运行状态。程序运行时则被定义为一个 State Monad。

```haskell
type ProgramState = (Program, State, Label)
type Runtime = Monad.State ProgramState
```

3.2 Computation

我们使用一个函数来定义某个快照的后继（下面忽略了辅助指令）：

```haskell
successor :: Program -> Snapshot -> Snapshot
successor program (i, State vars labels)
    | i == t = (t, s)
    | otherwise = case program !! i of
      Nop  -> (i + 1, State vars labels)
      Inc v -> (i + 1, State (Map.alter (\val -> Just (fromMaybe 0 val +
          1)) v vars) labels)
      Dec v -> (i + 1, State (Map.alter (\val -> Just (max (fromMaybe 0
          val - 1) 0)) v vars) labels)
      Gnz v l | vars ! v == 0  -> (i + 1, s)
              | notMember l labels -> (t, s)
              | labels ! l < 0  -> (t, s)
              | labels ! l >= t -> (t, s)
              | otherwise       -> (labels ! l, s)
    where s = State vars labels; t = length program
```

接下来，一个程序的**计算**被实现为从无限长的快照序列中取到终止为止：

```haskell
computation :: Program -> State -> [Snapshot] -- equivalent as "takeUntil"
computation p s = foldr (\x ys -> if fst x /= length p then x:ys else [x])
    [] $ iterate (successor p) (0, s)
```

3.3 ProgramState Monad

目前，已经实现了从一个指令序列（即程序）中持续计算得到结果的功能。但是这并没有协助我们方便地写出程序。

一个理想的状态是，我们可以像写汇编一样写指令序列，同时要能够使用具名变量，方便地设置标号，以及支持宏。

先来考虑程序和宏的类型性质。一个 Program 是指令的序列 [Instruction]，而一个宏可以被视为一个程序。那么，当宏插入一个程序时，应有性质：Program[Program] -> Program，也即插入宏的程序仍是一个程序。这种性质叫作 Flatten。

一个极为典型的满足 Flatten 性质的结构便是 Monad。因此，利用 Monad，我们可以很方便地将一段短程序组织成一个宏并复用。

更进一步，由于一个 $\mathscr{S}$ 程序显然地有一个状态，因此很容易想到利用 Haskell 的 State Monad 来管理程序的状态：

```
type ProgramState = (Program, State, Label)
type Runtime = Monad.State ProgramState
```

这便是此前所提到的程序运行环境定义。

Monad.State 接受一个 (s -> (a, s)) 类型的函数进行构造。首先，我们构造一个用于向程序添加指令的函数，这个函数接受一条指令，并返回一个 (s -> (a, s)) 形式的函数：

```
appendIr :: Instruction -> ProgramState -> (Address, ProgramState)
appendIr ir (p, s, e) = (length p, (p ++ [ir] , s, e))
```

然后，我们便可以实现一系列向程序中新增一条基本指令的函数：

```
nop :: Runtime Address
nop = state $ appendIr Nop

inc :: Variable -> Runtime Address
inc v = state $ appendIr $ Inc v

dec :: Variable -> Runtime Address
dec v = state $ appendIr $ Dec v

gnz :: Variable -> Label -> Runtime Address
gnz v l = state $ appendIr $ Gnz v l
```

接下来，我们定义用于控制标号的函数：

```haskell
_label_ :: Label -> Runtime Address
_label_ l = M.state $ \(p, State vs ls, e) ->
    let addr = length p
    in (addr, (p, State vs (Map.insert l addr ls), e))

_exit_ :: Label -> Runtime Address
_exit_ e = M.state $ \(p, State vs ls, _) -> (length p, (p, State vs ls, e))
```

当在某个指令的前面插入 _label_ a 后，便会将程序当前的指令地址存入标号 a 中。调用 _exit_ e 则会将 ProgramState 的 Exit 标号设为 e。

现在，利用 Haskell 的 do-notation，我们已经可以像汇编一样编写 $\mathscr{S}$ 语言程序了：

```haskell
almostId y x = do
    _label_ a
    dec x
    inc y
    gnz a
```

这个程序计算函数

$$f(x) = \begin{cases} 1 & \text{if } x = 0 \\ x & \text{otherwise} \end{cases}$$

3.4 Context

在可计算性理论中，变量的生成是任意的，每一次宏展开用一套新的变量并无问题。但是在实际的程序中，这会导致大量的只用过一次的变量存在，给调试带来了冗余信息，也造成了空间浪费。因此，重用变量是很有必要的。

为了重用临时变量，我们需要一个局部**上下文**的概念。在这个上下文中创建的变量，将会在出了上下文后被销毁。此后，新的临时变量仍可复用这些曾用过的变量的 ID，从而实现变量重用。此外，上下文还可以通过重写退出标号实现局部退出功能。

首先，我们构造用于生成若干个可用变量/标号的函数，以及一个取得当前程序地址的函数（略去实现）：

```haskell
freeVars :: Int -> Runtime [Variable]
freeLabels :: Int -> Runtime [Label]
curAddr :: Runtime Address
```

然后，一个上下文定义如下：

```
context :: Int -> Int -> (([Var], [Label]) -> Runtime Addr) -> Runtime Addr
context nVars nLabels program = do
    (_, State vars _, exit) <- M.get
    vs <- freeVars nVars
    (e:ls) <- freeLabels $ 1 + nLabels
    _exit_ e
    program (vs, ls)
    _label_ e
    (p, State _ labels, _) <- M.get
    M.put (p, State vars labels, exit)
    curAddr
```

该上下文完成了以下工作：

1. 创建了 nVars 个临时变量，nLabels 个自由标号，以及 1 个退出标号 e；
2. 将 e 设置为 program 的结束地址，并将当前上下文的退出标号设为 e；
3. 调用 program。该 program 可以自由使用新生成的变量与标号；
4. 将 ProgramState 的变量表与退出标号恢复到进入上下文之前的状态；
5. 返回结尾的指令地址。

注意上下文退出时，并不恢复标号至进入前的状态，这是因为标号的值持续有效，直至程序结束，不能重用。

现在，我们可以用 context 来创建具有上下文的程序了，样例程序将在下一节给出。

3.5 Basic Constant and Macros

为了统一起见，定义 ID 小于 0 的变量为保留常量：

```
true :: Variable
true = Var (-1) -- 1

false :: Variable
false = Var (-2) -- 0
```

这些常量的值将在程序环境初始化时送入。

现在，我们实现一些常用的宏：

goto 利用 true 常量，我们可以仅用一条指令完成 goto 动作。

```
goto :: Label -> Runtime Address
goto = gnz true
```

clr 利用 false 常量，也可以仅用一条指令完成变量清零动作。

```
clr :: Variable -> Runtime Address
clr v = mov v false
```

exit 直接跳转到当前上下文的结束标号。

```
exit :: Runtime Address
exit = M.get >>= \(_, _, exit) -> goto exit
```

gz 该宏用两条指令实现了 $IF\ Z = 0\ GOTO\ L$。

```
gz :: Variable -> Label -> Runtime Address
gz v l = context 0 1 $ \([], [e]) -> do
    gnz v e
    goto l
    _label_ e
```

4 Function

4.1 Definition

签名 一个函数的签名应由参数与返回值组成。具体到变量上，就是 n 个输入变量 $[Xs]$ 与 1 个输出变量 Y。

我们将 $Y \leftarrow f(X_1, X_2, ..., X_n)$ 的签名写成 $(Y, [X_1, X_2, ..., X_n])$，规定一个函数的计算结果会保存在变量 Y 中。由此，函数的签名定义如下：

```
type Signature = (Variable, [Variable])
```

函数 一个函数应能够接受与签名相合的变量列表，生成一串指令序列。此外，由于签名列表本身不带参数数量信息，因此有必要用额外字段指定。

因此，一个函数定义如下：

```
data Function = Function {
    argc :: Int,
    func :: Signature -> Runtime Address
}
```

接下来，构造一个创建函数调用上下文的函数：

```
function :: Int -> (Signature -> Runtime Address) -> Function
function argc func = Function argc $ \(out, args) -> context 0 0 $ \_ -> do
    (p, State vars labels, exit) <- M.get
    let vars' = Map.insertWith (const id) out 0 vars
        vars'' = foldl (\vs v -> Map.insertWith (const id) v 0 vs) vars' args
    M.put (p, State vars'' labels, exit)
    rest <- freeVars $ max (argc - length args) 0
    func (out, args ++ rest)
```

function 方法会保留 $(1 + \text{argc})$ 个变量空间，当实参过多时进行截断，实参不足时进行补零，用以代表输入变量与输出变量。

由此，我们可以定义 n 元函数的概念：

```
nullary  = function 0
unary    = function 1
binary   = function 2
ternary  = function 3
```

为了能在函数中使用局部变量与标号，我们将 function 与 context 结合起来使用：

```
functionC argc nLocals nLabels func
= function argc $ \(y, xs) -> context nLocals nLabels $ \(zs, ls) -> func
    (y, xs, zs, ls)
```

现在，可以通过 {x-naryC} 格式声明一个带局部变量与标号的函数：

```
someFunc = binaryC 1 2 $ (y, [x1, x2], [z1], [a, b]) -> (do ret)
```

4.2 Computation

我们构造一个计算函数的入口，这个入口会构造初始状态，并对函数进行计算：

```
computeFunction :: Function -> [Value] -> ([Snapshot], Program)
computeFunction (Function _ func) args =
    let inputs    = take (length args) (Var <$> [1..]) -- input starts from 1
        signature = (Var 0, inputs) -- goto -1 will terminate program
        emptyState = ([], State Map.empty Map.empty, Label (-1))
        (p, State vs ls, _) = M.execState (func signature) emptyState
        constants = [(Var 0, 0), (true, 1), (false, 0)]
        initVars  = Map.fromList $ zip inputs args ++ constants
        initState = State (Map.union initVars vs) ls -- feed state with
            inputs
    in (computation p initState, p)
```

接下来，我们通过一个函数正式计算一个 $\mathscr{S}$ 语言函数程序：

```
invoke :: Function -> [Value] -> Value -- inovke as true function
invoke func args =
    let (snapshots, _) = computeFunction func args
    in (varTable . snd . last $ snapshots) ! Var 0 -- 0 reserved for output
```

invoke 函数充当了 Haskell 与 $\mathscr{S}$ 语言之间的接口，使得 Haskell 能真正调用 $\mathscr{S}$ 语言的程序进行计算。现在，Haskell 的函数也可由 $\mathscr{S}$ 语言实现了。

4.3 Macros

我们实现两个与函数调用相关的宏。

call 一般的函数调用下，输入变量的值是可能被修改的。call 宏指令则创造了一个函数局部环境，将所有输出变量拷贝一份，并保证结果回送给输出变量。

```
call :: Function -> Signature -> Runtime Address
call (Function _ func) (out, ins)
    = context (1 + length ins) 0 $ \(y:xs, []) -> do
        clr y
        mapM_ (uncurry mov) $ zip xs ins
        func (y, xs)
        mov out y
```

ret 这个指令用于执行函数的返回动作，目前仅被实现为一个简单的 exit 指令。若在 ProgramStates 中记录输出变量的话，则可在 ret 中实现 return v 的效果。

```
ret :: Runtime Address
ret = exit
```

5 Primitive

5.1 Primitive Operators

本节用 $\mathscr{S}$ 语言实现了两个原始递归函数所使用的算子。

com 组合算子。

```
com :: Function -> [Function] -> Function
com f gs | (length gs == k) && all (\g -> argc g == n) (tail gs) =
    function n $ \(y, xs) ->
    context k 0 $ \(zs, []) -> do
        (_, State vars _, _) <- M.get
        mapM_ (\(g, z) -> call g (z, xs)) $ zip gs zs
        call f (y, zs)
    where k = argc f
          n = argc $ head gs
```

rec 递归算子。

```
rec :: Function -> Function -> Function
rec f g | argc g == n + 2 = function (n + 1) $ \(y, t:xs) ->
    context 1 1 $ \([z], [a]) -> do
        (_, _, exit) <- M.get
        call f (y, xs)
        _label_ a
        gz t exit
        call g (y, z:y:xs)
        inc z
        dec t
        goto a
    where n = argc f
```

5.2 Initial Functions

本节用 $\mathscr{S}$ 语言实现三个原始递归函数所使用的初始函数。

zero 零元 - 零函数。

```
z :: Function
z = nullary $ \(y, _) -> mov y false
```

successor 一元 - 后继函数。

```
s :: Function
s = unary $ \(y, [x]) -> inc x >> mov y x
```

projection n 元 - 投影函数。

```
u :: Int -> Int -> Function
u n i = function n $ \(y, xs) -> mov y $ xs !! i
```

5.3 Basic Function and Operators

本节利用 PRC class 实现一些用于编写原始递归函数的重要工具函数。

identity 利用投影实现的更为简单的恒等函数。

```
id' :: Function
id' = u 1 0
```

iota 对函数 f 进行 n 次迭代调用。

```
iota :: Function -> Function -- Nth iteration of f
iota f = rec id' (com f [u (2 + argc f) 1])

(>^<) :: Function -> Value -> Function
f >^< n = unary $ \(y, [x]) -> do
    set y n
    call (iota f) (y, [y, x])
```

constant 常函数，由零函数进行 n 次后继函数迭代而得。

```
k :: Value -> Function -- Constant k
k n = com (s >^< n) [z]
```

% 重写函数的参数个数。一个常用的用法是重写零元常函数的参数个数，使其对任意输入都返回常值。

```
(%) :: Function -> Int -> Function -- rewrite argc
(Function _ f) % n = Function n f
```

reverse 反转参数列表。用于使函数调用更符合一般习惯。

```
rev :: Function -> Function -- Flip argument list
rev f | argc f == 2 = com f [u 2 1, u 2 0]
```

apply 函数柯里化。接受一个函数与一个参数，并返回一个接受剩下参数的函数。

```
apply :: Function -> Value -> Function -- Function currying
apply f a = let n' = (argc f - 1); p = u n'
         in com f $ (k a % n'):map p [0..n'-1]
```

pass 从 n 元函数中构造 $n+k$ 元函数，其中前 k 个参数在调用时被忽略。

```
pass :: Function -> Int -> Function -- Pass first k parameters
pass f k = let n = argc f; p = u (k + n)
         in com f (map p [k..k+n-1])
```

fold 折叠运算，也即原始递归中的循环运算实现。

```
fold :: Function -> Function -> Function -- Fold a function by its first
    parameter rolling
fold b f | argc b == 2 =
    let n = argc f; p = u (n + 1)
    in rec (apply f 0) (com b [p 1, com f (com s [p 0]:map p [2..n])])
```

Part III

Application

6 Mini Programs

mov Mov 指令的基本指令宏实现。

```
mov y x = context 1 5 $ \([z], [a, b, c, d, e]) -> do
    clr y
    _label_ a
    gnz x b
    goto c
    _label_ b
    dec x
    inc y
    inc z
    goto a
    _label_ c
    gnz z d
    goto e
    _label_ d
    dec z
    inc x
    goto c
    _label_ e
```

id

```
identity = unary $ \(y, [x]) -> context 1 2 $ \([z], [a, b]) -> do
    _label_ a
    gnz x b
    exit
    _label_ b
    dec x
    inc y
    gnz x a
```

这个程序计算函数 $f(x) = x$.

add

```
add = binaryC 1 2 $ \(y, [x1, x2], [z], [a, b]) -> do
    mov y x1
    mov z x2
    _label_ b
    gnz z a
    exit
    _label_ a
    dec z
    inc y
    goto b
```

这个程序计算函数 $f(x, y) = x + y$.

threefold

```
triple = unary $ \(y, [x]) -> do
    call add (y, [x, x])
    call add (y, [y, x])
```

这个程序计算函数 $f(x) = 3x$.

7 Some PRC Functions

7.1 Arithmetic

前驱函数

```
pre = rec z (u 2 0)
```

该程序计算了函数

$$f(x) = \begin{cases} 0 & \text{if } x = 0 \\ x - 1 & \text{otherwise} \end{cases}$$

加法

```
add = rec (u 1 0) (com s [u 3 1])
```

乘法/阶乘/幂

```
mul = rec (z % 1) (com add [u 3 1, u 3 2])
fac = rec (k 1) (com mul [com s [u 2 0], u 2 1])
pow = rev $ rec (k 1 % 1) (com mul [u 3 1, u 3 2])
```

减法（溢出则返回 0）

```
sub = rev $ rec (u 1 0) (com pre [u 3 1])
```

绝对值距离

```
abf = com add [sub, rev sub]
```

7.2 Logic and Relation

取反

```
inv = com sub [k 1 % 1, u 1 0]
```

符号函数

```
sgn = com inv [inv]
```

且/或

```
or' = com sgn [add]
and' = com sgn [mul]
```

等于/不大于/不大于/小于/大于

```
eq = com inv [abf]
le = com inv [sub]
ge = rev le
lt = com inv [ge]
gt = com inv [le]
```

7.3 Condition and Iteration

if then else

```
if' p t f = com add [com mul [p, t], com mul [com inv [p], f]]
```

求和/连乘/存在/对任意

```
sum' = fold add -- sum' id': 等差数列求和
prod = fold mul -- prod (com s [id']): 阶乘
any' f = com sgn [sum' f]
all' f = com sgn [prod f]
```

7.4 Miscellaneous

最小值

```
min' p = if' (any' p) (sum' $ prod $ com inv [p]) (z % argc p)
```

整除/取余

```
div' = com (min' $ com gt [com mul [com s [u 3 0], u 3 1], u 3 2]) [u 2 0, u 2
    1, u 2 0]
mod' = com sub [u 2 0, com mul [u 2 1, div']]
```

8 QuickSort

囿于时间所限，本报告未能实现到 Sequence Encoding 及 Universal Programs 章节。尽管如此，本节尝试用已实现的 $\mathscr{S}$ 语言原始递归语法，给出排序问题的一种解决方案。

本报告选取的方案是快速排序。我们先来看一个典型的 Haskell 实现：

```
quickSort :: (Ord a) => [a] -> [a]
quickSort [] = []
quickSort (x:xs) =
    let smaller = [a | a <- xs, a <= x]
        greater = [a | a <- xs, a > x]
    in (quickSort smaller) ++ [x] ++ (quickSort greater)
```

可以看到，想要利用 $\mathscr{S}$ 语言的原始递归机制实现快速排序，我们需要准备以下工具：head, tail, makeSeq, append, concat, filter。

我们来逐一实现这些概念。

head & tail 获取哥德尔序列的头部数与尾序列。

```
head' = con lookup' [id', z % 1] -- lookup': retrieve by index
tail' = con div' [id', head']
```

makeSeq 将一个数包装成一个最简单的哥德尔序列。这十分简单：

```
makeSeq = com pow [k 2 % 1, id'] -- 2^x
```

append & concat 将一个数添加到哥德尔序列末尾，以及拼接两个哥德尔序列。

```
append = com mul [id', com pow [com prime [length'], u 2 1]]
concat' = fold append $ rec (u 2 0) (com lookup' [u 3 2, u 3 0])
```

filter 通过一个 Predicate 过滤哥德尔序列。

```
filter' p =
    let proc = if' p append (id' % 2) -- append or stay the same
        iter = rec (z % 1) (com lookup' [u 3 2, u 3 0])
    in com (fold proc iter) [len, id'] -- roll by index
```

最终，我们将这些函数拼装起来，从而实现快速排序函数：

```
baseCase = z % 1 -- zero denotes empty list []

recStep = -- u 3 1: iterating x; u 3 2: the pivot
    let smaller = com (filter' $ com le [u 3 1, u 3 2] % 2) [tail', head']
        greater = com (filter' $ com gt [u 3 1, u 3 2] % 2) [tail', head']
        smaller' = com quickSort [smaller]
        greater' = com quickSort [greater]
    in com concat' [com concat' [smaller', com makeSeq [head']], greater']

quickSort = if' (con eq [id', z % 1]) baseCase recStep
```