第U章 ちっちゃな命令型言語 Util

U.1 Util コンパイラー

この節では、Util コンパイラーの実装を紹介していく。

実際のコンパイラーにはフロントエンド、つまり　　や　　が必要である。字句解析や構文解析の原理は Haskell でも C 言語などの命令型言語で記述するときと変わりはない。再帰下降構文解析法、あるいは LR 構文解析法などの方法を利用する。再帰下降法で構文解析部を記述するときには、後述のようにモナドを利用することができる。

しかし、ここではこれらフロントエンドの作り方は既知のものとして、構文木ができた状態から話をはじめることにする。

Util の構文木のデータ構造として、次のような Haskell のデータ型を使用する。

ファイルRecType.hs

```
type Decl = (String, Expr)
data Expr = Const Target       -- 定数（Target は後述）
          | Var String           -- 変数
          | If Expr Expr Expr    -- if 文
          | While Expr Expr      -- while 文
          | Begin [Expr]         -- ブロック
          | Let [Decl] Expr      -- let 式（関数定義）
          | Val Decl Expr        -- val 式（変数定義）
          | Lambda String Expr   -- ラムダ式
          | Delay Expr           -- delay 式（後述）
          | App Expr Expr        -- 関数適用
         deriving Show
```

つまり、式 (Expr)とは、定数 (Const) または、変数 (Var) または、 **if** 式 (If)、 **let** 式 (Let)、ラムダ式 (Lambda)、関数適用 (App)などからなる。（あとから必要に応じて構文要素を追加することにする。）

データ型 Expr の定義は、Util の抽象構文を代数的データ型として直訳したものである。

U.1.1 字句解析・構文解析関数

次のような関数が既に定義されているものと仮定する。

```
myParse :: String -> Expr    -- 字句解析・構文解析の関数
```

- "**val** x <- 2 * 2 **in val** y <- x * x **in** y * y" というソースプログラムは Expr 型のデータとして次のように構文解析される。

```
Val ("x", (App (App times (Const (TLit (Int 2))))
               (Const (TLit (Int 2)))))
    (Val ("y", (App (App times (Var "x")) (Var "x")))
               (App (App times (Var "y")) (Var "y")))
```

  ただし、*times* は * に対応する Expr の式である。

- "\ f x -> f x" という式は、

```
Lambda "f" (Lambda "x" (App (Var "f") (Var "x")))
```

  というデータに構文解析される。

- 「&&」,「||」は、それぞれ、

  b1 && b2 ↦ **if** b1 **then** b2 **else** False
  b1 || b2 ↦ **if** b1 **then** True **else** b2

  という糖衣構文であるように構文解析されるようにしておく。

U.1.2 ターゲット言語

コンパイラーのターゲット言語である Haskell のサブセットの構文木を表現する型 Target 型を定義しておく。

ファイルTarget.hs

```haskell
type TDecl = (String, Target)
data Target = TLit Literal              -- 定数
            | TVar String               -- 変数
            | TIf Target Target Target -- if 文
            | TLet [TDecl] Target       -- let 文
            | TLambda1 String Target    -- ラムダ式
            | TApp1 Target Target       -- 関数適用
            | TReturn Target            -- return に相当
            | TBind Target Target       -- (>>=) に相当
           deriving (Show,Eq)
data Literal = Str String | Int Integer
             | Frac Rational | Char Char
             deriving (Show,Eq)
```

Util のコンパイラーとは次のような型を持つ関数である。

```
comp :: Expr -> Target    -- コンパイラー
```

U.1.3 コンパイラーの定義

関数 comp の定義は次のようになる。個々の構文要素に対する定義は比較的素直である。

ファイルRecCompiler.hs

```
comp :: Expr -> Target
comp (Const c)       = TReturn c
comp (Var x)         = TReturn (TVar x)
comp (Val (x, m) n) = comp m `TBind` TLambda1 x
                       (comp n)
comp (Let decls n)   = TLet (map (\ (x, m) ->
                          let TReturn c = comp m
                            in (PVar x, c)) decls)
                           (comp n)
comp (App f x)       = comp f `TBind` TLambda1 "_f"
                      (comp x `TBind` TLambda1 "_x"
                      (TApp1 (TVar "_f") (TVar "_x")))
comp (Lambda x m)    = TReturn (TLambda1 x (comp m))
comp (Delay m)       = TReturn (comp m)
comp (If e1 e2 e3)   = comp e1 `TBind` TLambda1 "_b"
                       (TIf (TVar "_b")
                            (comp e2) (comp e3))
comp (While e1 e2)   = TLet [(PVar "_while",  body)]
                            (TVar "_while")
   where body = comp e1 `TBind` TLambda1 "_b"
                (TIf (TVar "_b")
                     (comp e2 `TBind` TLambda1
                                      (TVar "_while"))
                     (TReturn (TVar "()")))
comp (Begin [e])     = comp e
comp (Begin (e:es)) = comp e `TBind` TLambda1
                        (TVar "_") (comp (Begin es))
```

右辺で使われている `_f`, `_x`, `_b`, `_while` などの識別子は、Util ソースプログラム中に使われている識別子と衝突しないように選んでいる。

U.2 UtilCont — 接続の導入

Util に **break**, **continue** などを導入するために、`Expr` の定義に次のように構成子を追加する。また、**goto** 文 を導入するため、ラベルも導入する。
ファイル ContType.hs

```
data Expr = Const Target | Var String
          | If Expr Expr Expr | While Expr Expr
          | Let [Decl] Expr | Val Decl Expr
          | Lambda String Expr | Delay Expr
          | App Expr Expr
          -- ここまでは、Util1と同じ
          | Begin [LabeledExpr]      -- ブロック
          | Break                    -- break 文
          | Continue                 -- continue 文
          | Goto String              -- goto 文
          deriving Show
type LabeledExpr = (Maybe String, Expr) -- ラベル付き式
```

`Const`, `Var`, `Let` などに対しては `comp` は変更する必要はない。変更された部分のうち、`Goto`, `Break`, `Continue` に対する `comp` の定義は以下のようになる。
ファイル ContCompiler.hs

```
comp (Goto lbl)         = mkGoto lbl "()"
comp Break              = mkGoto "_break" "()"
comp Continue           = mkGoto "_while" "_break"
```

```
4
5  mkGoto lbl v = TApp1 (TVar "abort")
6                       (TApp1 (TVar lbl) (TVar v))
```

また、`While` に対する `comp` の定義は次のようになる。

ファイル ContCompiler.hs

```
1  comp (While e1 e2)      = compWhile e1 e2
2
3  compWhile e1 e2 = TApp1 (TVar "KIO")
4    (TLambda1 "_break"
5      (TLet [(PVar "_while", TApp1 (TVar "unKIO") body)]
6            (TApp1 (TVar "_while") (TVar "_break"))))
7    where body = comp e1 `TBind` TLambda1 "_b"
8            (TIf (TVar "_b") (comp e2 `TBind`
9                   TLambda1 "_"
10                  (TApp1 (TVar "KIO") (TVar "_while")))
11                  (TReturn (TVar "()")))
```