ФП и трансляция программ

Трансляция — преобразование программы, представленной на исходном языке, в эквивалентную ей программу на целевом языке:

$$P_{in},\ T\ \rightarrow\ P_{out}$$

где T — трансформационная грамматика:

$$T = (V_{in},\ V_{out},\ V_{non-term},\ S,\ R),$$

где $V_{in},\ V_{out}$ — алфавиты исходного и целевого языков, $V_{non-term}$ — алфавит нетерминальных символов[1], S — стартовый («корневой») нетерминальный символ, R — правила трансляции.

По схеме:

программа на исходном языке $\xrightarrow{\text{parser}}$ промежуточное представление (дерево) $\xrightarrow{\text{generator}}$ программа на целевом языке

[1] Т. е. последовательностей символов, обозначающих сущности языка, не имеющих конкретного символьного выражения

Пример

Forth $\longrightarrow$ Haskell

Грамматика

```
программа := {пробел}, { строчный_комментарий
                       | строка_для_печати
                       | целое
                       | слово}, {пробел};
строка_для_печати := '."', пробел, NOT '"', '"', конец_слова;
строчный_комментарий := '\', пробел, ..., '\n', {пробел};
целое := ('-', цифра) | цифра, {цифра}, конец_слова;
слово := !пробел, {!пробел}, конец_слова;
конец_слова := пробел, {пробел};
пробел := ' ' | '\t' | '\n' | '\r' | '\f' | '\v'.
```

Начало и конец стековых комментариев, определений новых слов, условных конструкций, циклов являются словами. Не показан запрет на размещение блочных конструкций вне определений новых слов.

Грамматика для построения дерева

```
программа := {пробел}, { определение
                       | строчный_комментарий
                       | стековый_комментарий
                       | строка_для_печати
                       | целое
                       | слово}, {пробел};
определение := ':', имя,  слово*, {слово*}, ';', конец_слова;
имя := !пробел, {!пробел}, конец_слова;
управляющая_конструкция := НАЧАЛО, конец-слова, слово*, {слово*}, КОНЕЦ, конец-слова;
строка_для_печати := '."', пробел, !'"', '"', конец_слова;
строчный_комментарий := '(', пробел, ...,'\n', {пробел};
стековый_комментарий := '(', пробел, ..., конец_слова, ')', конец_слова;
целое := ('-', цифра) | цифра, {цифра}, конец_слова;
слово := !пробел, {!пробел}, конец_слова;
конец_слова := пробел, {пробел};
```

`слово*` — слово или управляющая конструкция, применение которой разрешено в пределах объемлющей управляющей конструкции или определения (например, `leave` внутри `do-loop`), включая строки для печати, строчные и стековые комментарии.

Дерево

```
-- fs2hs.hs

data W = IntNum    Int
    | Word         String
    | Quoted       String
    | StackComment String
    | LineComment  String
    | Definition   String [W]
    | IfThen       [W]
    | DoLoop       [W]
    | DoLoopPlus   [W]
    | BeginUntil   [W]
    deriving (Show, Read, Eq)
```

Программа на Форте: «корневой» уровень

```
-- fs2hs.hs

import Text.ParserCombinators.Parsec

program = many1 ( choice [ try definition
                         , try int
                         , try quoted
                         , try stackComment
                         , try lineComment
                         , word
                         ] )
```

Слово

```
-- fs2hs.hs

word :: Parser W
word = do
    w <- many1 $ noneOf spcs
    endOfWord
    return $ Word w

spcs = " \t\n\r\f\v"

endOfWord = eof <|> (oneOf spcs >> spaces)
```

Целое число

```
-- fs2hs.hs

int :: Parser W
int = uint <|> sint

uint :: Parser W
uint = do
    ds <- many1 $ digit
    endOfWord
    return $ IntNum $ read ds

sint :: Parser W
sint = do
    char '-'
    ds <- many1 $ digit
    endOfWord
    return $ IntNum $ read $ "-" ++ ds
```

Строка в кавычках для вывода в консоль

```
-- fs2hs.hs

quoted :: Parser W
quoted = do
    char '.'
    char '"'
    oneOf spcs
    q <- manyTill anyChar (try (char '"' >> endOfWord))
    return $ Quoted q
```

Комментарии

```
-- fs2hs.hs

stackComment :: Parser W
stackComment = do
    char '('
    endOfWord
    c <- manyTill anyChar (try ( oneOf spcs >> char ')' >> endOfWord))
    return $ StackComment c

lineComment :: Parser W
lineComment = do
    char '\\'
    endOfWord
    c <- manyTill anyChar (try (char '\n' >> many (oneOf spcs)))
    return $ LineComment c
```

Определение нового слова

```
-- fs2hs.hs

definition :: Parser W
definition = do
    char ':'
    endOfWord
    name <- many1 $ noneOf spcs
    endOfWord
    ws <- manyTill ( choice [ try quoted
                            , try int
                            , try stackComment
                            , try lineComment
                            , try ifThen
                            , try doLoopPlus
                            , try doLoop
                            , try beginUntil
                            , word
                            ]
                   ) (try (char ';' >> endOfWord))
    return $ Definition name ws
```

Условная конструкция `if ... then`

```
-- fs2hs.hs

ifThen :: Parser W
ifThen = do
    string "if"
    endOfWord
    ws <- manyTill ( choice [ try quoted
                            , try int
                            , try stackComment
                            , try lineComment
                            , try ifThen
                            , try doLoopPlus
                            , try doLoop
                            , try beginUntil
                            , word
                            ]
                   ) (try (string "then" >> endOfWord))
    return $ IfThen ws
```

Цикл с параметром `do ... loop`

```
-- fs2hs.hs

doLoop :: Parser W
doLoop = do
    string "do"
    endOfWord
    ws <- manyTill ( choice [ try quoted
                            , try int
                            , try stackComment
                            , try lineComment
                            , try ifThen
                            , try doLoopPlus
                            , try doLoop
                            , try beginUntil
                            , word
                            ]
                   ) (try (string "loop" >> endOfWord))
    return $ DoLoop ws
```

Цикл с параметром `do ... +loop`

```
-- fs2hs.hs

doLoopPlus :: Parser W
doLoopPlus = do
    string "do"
    endOfWord
    ws <- manyTill ( choice [ try quoted
                            , try int
                            , try stackComment
                            , try lineComment
                            , try ifThen
                            , try doLoopPlus
                            , try doLoop
                            , try beginUntil
                            , word
                            ]
                   ) (try (string "+loop" >> endOfWord))
    return $ DoLoopPlus ws
```

Цикл с постусловием `begin ... until`

```
-- fs2hs.hs

beginUntil :: Parser W
beginUntil = do
    string "begin"
    endOfWord
    ws <- manyTill ( choice [ try quoted
                            , try int
                            , try stackComment
                            , try lineComment
                            , try ifThen
                            , try doLoopPlus
                            , try doLoop
                            , try beginUntil
                            , word
                            ]
                   ) (try (string "until" >> endOfWord))
    return $ BeginUntil ws
```

Наглядный вывод дерева

```
-- fs2hs.hs

printTree :: String -> [W] -> IO ()

printTree _ [] = return ()

printTree indent ((Definition name inners):outers) = putStr indent
    >> putStr "Definition "
    >> print name
    >> printTree (indent ++ "\t") inners
    >> printTree  indent outers

printTree indent ((DoLoop inners):outers) = putStr indent
    >> putStrLn "DoLoop"
    >> printTree (indent ++ "\t") inners
    >> printTree  indent outers

-- см. след. стр.
```

Наглядный вывод дерева (окончание)

```
-- fs2hs.hs
-- printTree -- окончание

printTree indent ((DoLoopPlus inners):outers) = putStr indent
    >> putStrLn "DoLoopPlus "
    >> printTree (indent ++ "\t") inners
    >> printTree  indent outers

printTree indent ((BeginUntil inners):outers) = putStr indent
    >> putStrLn "BeginUntil "
    >> printTree (indent ++ "\t") inners
    >> printTree  indent outers

printTree indent ((IfThen inners):outers) = putStr indent
    >> putStrLn "IfThen "
    >> printTree (indent ++ "\t") inners
    >> printTree  indent outers

printTree indent (x:xs) = putStr indent
    >> print x
    >> printTree indent xs
```

Функция для тестирования разбора

```
-- fs2hs.hs

test = do
    src <- readFile "tree-demo.fs"
    case parse program "" src of
        Left  err -> print err
        Right ws  -> printTree "" ws
```

Пример дерева

```
\ tree-demo.fs

: sqr ( x -- x*x )
    dup *
;

: signum ( x -- -1|0|1 )
    dup 0 > if drop  1 then
    dup 0 = if drop  0 then
    dup 0 < if drop -1 then
;

."  9 sqr: " 9 sqr . cr
." -5 signum: " -5 signum . cr
```

```
ghci> test
LineComment "tree-demo.fs"
Definition "sqr"
    StackComment "x -- x*x"
    Word "dup"
    Word "*"
```

Пример дерева (продолжение)

```
\ tree-demo.fs

: sqr ( x -- x*x )
    dup *
;

: signum ( x -- -1|0|1 )
    dup 0 > if drop  1 then
    dup 0 = if drop  0 then
    dup 0 < if drop -1 then
;

."  9 sqr: " 9 sqr . cr
." -5 signum: " -5 signum . cr
```

```
Definition "signum"
    StackComment "x -- -1|0|1"
    Word "dup"
    IntNum 0
    Word ">"
    IfThen
        Word "drop"
        IntNum 1
    Word "dup"
    IntNum 0
    Word "="
    IfThen
        Word "drop"
        IntNum 0
    Word "dup"
    IntNum 0
    Word "<"
    IfThen
        Word "drop"
        IntNum (-1)
```

Пример дерева (окончание)

```
\ tree-demo.fs

: sqr ( x -- x*x )
    dup *
;

: signum ( x -- -1|0|1 )
    dup 0 > if drop  1 then
    dup 0 = if drop  0 then
    dup 0 < if drop -1 then
;

."  9 sqr: " 9 sqr . cr
." -5 signum: " -5 signum . cr
```

```
Quoted " 9 sqr: "
IntNum 9
Word "sqr"
Word "."
Word "cr"
Quoted "-5 signum: "
IntNum (-5)
Word "signum"
Word "."
Word "cr"
```

Слова Forth $\Leftrightarrow$ функции Haskell

Пусть $\mathbf{x}$ — стек, $\mathbf{x}_i$ — состояние стека, $\mathbf{x}_i = f_i(\mathbf{x}_{i-1})$ — функция, эквивалентная слову Forth. Тогда программа:

$$\mathbf{x}_1 = f_1(\mathbf{x}_0),\ \mathbf{x}_2 = f_2(\mathbf{x}_1)\ \cdots\ \mathbf{x}_{n-1} = f_{n-1}(\mathbf{x}_{n-1}),\ \mathbf{x}_n = f_n(\mathbf{x}_n).$$

$$\Updownarrow$$

$$g = f_n\ \circ\ f_{n-1}\ \circ\ \cdots\ \circ\ f_2\ \circ\ f_1,$$

$$\mathbf{x}_n = g(\mathbf{x}_0).$$

Слова Forth $\Leftrightarrow$ функции Haskell

Помимо стека, функции могут принимать другие аргументы, например:

$$f(\mathbf{x},\ x),$$

$$g = \cdots \circ\ f(x)\ \circ\ \cdots,$$

$$\mathbf{x}_n = g(\mathbf{x}_0).$$

Управляющая конструкция:

$$\mathbf{x}_n = h(g(\mathbf{x}_0)).$$

Цикл, доступ к параметру цикла:

$$g = \cdots \circ\ h(i \mapsto g(i))\ \circ\ \cdots.$$

Выборка «чистых» определений

```
-- fs2hs.hs

pass1 :: [W] -> String
pass1 [] = "\n"
pass1 ((Definition name ws):rest) =
    transDefinition name ws ++ pass1 rest
pass1 (w:ws) = pass1 ws -- ignore word
```

Трансляция определения нового слова

```
-- fs2hs.hs

transDefinition :: String -> [W] -> String
transDefinition name ws = name
    ++ " = "
    ++ transComposition ws
    ++ "\n"
```

Представление последовательности слов композицией функций

```
-- fs2hs.hs

transComposition :: [W] -> String
transComposition ws =
    foldl sep (head ts) (tail ts)
    where
        ts = reverse $ map transWord (filter inUse ws)
        sep w1 w2 = w1 ++ " . " ++ w2

inUse :: W -> Bool
inUse (StackComment _) = False
inUse (LineComment _) = False
inUse (Quoted _) = False
inUse (Word ".") = False
inUse (Word ".s") = False
inUse _ = True
```

Трансляция слов

```
-- fs2hs.hs

transWord :: W -> String

-- TODO: сюда будем вставлять реализации трансляции слов

transWord (Word w) = w  -- most universal
transWord x =  error $ "failed: " ++ x
```

Трансляция слов: арифметические операции

```
-- fs2hs.hs

transWord (Word "+") = "add"
transWord (Word "-") = "sub"
transWord (Word "*") = "mul"
transWord (Word "/") = "div'"
```

```
-- lib.hs

-- :: [Int] -> [Int]

add  (x2:x1:xs) = (x1 + x2):xs
sub  (x2:x1:xs) = (x1 - x2):xs
mul  (x2:x1:xs) = (x1 * x2):xs
div' (x2:x1:xs) = (x1 `div` x2):xs
```

Трансляция слов: операции сравнения

```
-- fs2hs.fs

transWord (Word "<" ) = "lt"
transWord (Word ">" ) = "gt"
transWord (Word "<=") = "le"
transWord (Word ">=") = "ge"
transWord (Word "=" ) = "eq"
transWord (Word "<>") = "ne"
```

```
-- lib.fs

lt (x2:x1:xs) = bool2int (x1 <  x2) : xs
gt (x2:x1:xs) = bool2int (x1 >  x2) : xs
le (x2:x1:xs) = bool2int (x1 <= x2) : xs
ge (x2:x1:xs) = bool2int (x1 >= x2) : xs
eq (x2:x1:xs) = bool2int (x1 == x2) : xs
ne (x2:x1:xs) = bool2int (x1 /= x2) : xs

bool2int True  = -1
bool2int False =  0
```

Трансляция слов: логические операции

```
-- fs2hs.fs

transWord (Word "or") = "or'"
transWord (Word "and") = "and'"
transWord (Word "not") = "not'"
```

```
-- lib.fs
or'  (x2:x1:xs) = (abs x1 + abs x2):xs
and' (x2:x1:xs) = (x1 * x2):xs

not' (0:xs) = (-1:xs)
not' (_:xs) = ( 0:xs)
```

Трансляция слов: операции со стеком

```
-- fs2hs.fs

transWord (Word "drop") = "drop'"
transWord (Word "2dup") = "dup2"
transWord (Word "2over") = "over2"

...
```

```
-- lib.fs

dup   (x:xs) = (x:x:xs)
drop' (x:xs) = xs

swap (x2:x1:xs) = (x1:x2:xs)
over (x2:x1:xs) = (x1:x2:x1:xs)

rot (x3:x2:x1:xs) = (x1:x3:x2:xs)

dup2  (x2:x1:xs) = (x2:x1:x2:x1:xs)
over2 (x4:x3:x2:x1:xs) = (x2:x1:x3:x3:x2:x1:xs)
```

Трансляция слов: целое — на вершину стека

```
-- fs2hs.fs

transWord (IntNum n) = "push " ++ show n
```

```
-- lib.fs

push x xs = x:xs
```

Главная функция транслятора

```
-- fs2hs.hs

main = do
    args <- getArgs
    src  <- readFile $ args !! 0
    lib  <- readFile  "lib.hs"
    case (parse program "" src) of
        Left  err -> print err
        Right ws  -> printTree "" ws
                     >> writeFile  (args !! 1) (fs2hs "" ws)
                     >> appendFile (args !! 1)  lib
```

Это уже можно использовать

```
ghci> :l fs2hs.hs
ghci> :main test.fs test.hs

(Дерево)

ghci> :l test
ghci> neg [1]
[-1]
```

```
\ test.fs

: neg 0 swap - ;
```

```
-- test.hs

neg = sub . swap . push 0

-- Далее следуют библиотечные функции
```

Реализация условий

```
-- fs2hs.fs

transWord (IfThen ws) = "ifThen ( "
    ++ transComposition ws
    ++ " )"
```

```
-- lib.fs

ifThen fn (flag:xs)
    | flag /= 0 = fn xs
    | otherwise = xs
```

Реализация циклов с постусловием

```
-- fs2hs.fs

transWord (BeginUntil ws) = "until1 ( "
    ++ transComposition ws
    ++ " )"
```

```
-- lib.fs

until1 fn xs = until2 fn (fn xs)

until2 fn (x:xs)
    | x == 0 = until2 fn (fn xs)
    | otherwise = xs
```

Реализация циклов с параметром

```
-- fs2hs.fs

transWord (DoLoop ws) = "doLoop (\\ i -> "
    ++ transComposition ws
    ++ " )"

transWord (Word "I") = "push i" -- in loops
transWord (Word "i") = transWord (Word "I") -- synonym for I
```

```
-- lib.fs

doLoop fn (from:upto:xs) = doLoop' fn from upto xs

doLoop' fn from upto xs
    | from < upto = doLoop' fn (from+1) upto (fn from xs)
    | otherwise = xs
```

Реализация циклов с параметром (продолжение)

```
-- fs2hs.fs

transWord (DoLoopPlus ws) = "doLoopPlus (\\ i -> "
    ++ transComposition ws
    ++ " )"
```

```
-- lib.fs

doLoopPlus fn (from:upto:xs) = doLoopPlus' fn from upto (fn from xs)

doLoopPlus' fn from upto (step:xs)
    | from' < upto = doLoopPlus' fn from' upto (fn from' xs)
    | otherwise = xs
    where
        from' = from + step
```

Пример: факториал

```
\ test.fs

: factorial  ( x -- x! )
    1 +      ( x+1 -- )
    1        ( x+1 1 -- )
    swap     ( 1 x+1 -- )
    1        ( 1 x+1 1 -- )
    do           ( 1 -- )
        I        ( 1 index -- )
        *        ( 1*index -- )
    loop ( factorial )
;
```

```
-- test.hs

factorial = doLoop (\ i -> mul . push i ) . push 1 . swap . push 1 . add . push 1
```

```
ghci> factorial [5]
[120]
ghci> factorial [10]
[3628800]
```

```
: sqrti ( n0 )
    dup 0 <  if drop then ( cause an error )
    dup 0 >= if ( n0 )
        1 1 ( n0 c0 d0 )
        rot ( c0 d0 n0 )
        begin         ( c0 d0 n0 )
            rot       ( d0 n0 c0 )
            1 +       ( d0 n0 c1 )
            rot       ( n0 c1 d0 )
            rot       ( c1 d0 n0 )
            over      ( c1 d0 n0 d0 )
            -         ( c1 d0 n1 )
            swap      ( c1 n1 d0 )
            2 +       ( c1 n1 d1 )
            2dup      ( c1 n1 d1 n1 d1 )
            <         ( c1 n1 d1 f1 )
            rot       ( c1 d1 f1 n1 )
            dup 0 =   ( c1 d1 f1 n1 f2 )
            rot       ( c1 d1 n1 f2 f1 )
            or        ( c1 d1 n1 f )
        until         ( c1 d1 n1 )
        drop drop     ( c1 )
        1 -           ( c0 )
    then
;
```

Пример: квадратный корень

```
sqrti =
    ifThen ( sub . push 1
           . drop' . drop'
           . until' ( or'
                    . rot
                    . eq . push 0 . dup
                    . rot
                    . lt
                    . dup2
                    . add . push 2
                    . swap
                    . sub . over . rot . rot
                    . add . push 1 . rot
                    )
           . rot . push 1 . push 1
           )
    . ge . push 0 . dup
    . ifThen ( drop' ) . lt . push 0 . dup
```