УДК 519.682.1
DOI: 10.18523/2617-3808.2022.5.4-11

Проценко В. С.

СПЕЦИФІКАЦІЯ ПРОЦЕДУРНОЇ МОВИ ПРОГРАМУВАННЯ

Розглянуто процедурну мову програмування, об'єкти якої – цілі змінні й процедури. Оператори мови – присвоєння, введення, виведення, умовний, циклу і блок. Головне призначення блоку – введення локальних цілих змінних і процедур. Процедура має параметри і тіло – оператор. Обчислює процедуру оператор виклику, аргументи якого цілі змінні. Наведено повну формальну специфікацію мови. На основі специфікації побудовано інтерпретатор мови програмування.

Ключові слова: мова програмування, програма, процедура, оператор, вираз, синтаксис, денотат, семантична функція, Haskell, синтаксичний аналіз, інтерпретатор.

Огляд мови

Розглянуто просту процедурну мову програмування, кожна програма якої може вводити цілі значення, обробляти їх і виводити нові цілі значення, як результат. Програма – окремий оператор, як правило, блок з описом локальних цілих змінних і процедур і списком операторів – тілом блоку. Допустимим є блок, який має тільки тіло – він еквівалентний списку операторів. Головне оброблення даних виконують оператори присвоювання v := e, введення **read** v, виведення **write** e, умовний **if** (e) s, циклу while (e) s, де v – змінна, e – вираз, s – оператор. В операторах умовному і циклу значення iv>0 цілого виразу е еквівалентно логічному значенню True. Призначення оператору блоку { **int** v1, ..., vn; p1; ... pk; s1;...; sm} ввести локальні цілі змінні v1, ..., vn і процедури p1, ..., pk, які можна вживати в операторах s1, ..., sm і процедурах p1, ..., pk. Процедура **proc** n (v1,..., vt) s має ім'я n, список, можливо порожній, формальних параметрів v1,..., vt і тіло s, яким є довільний оператор (найчастіше блок). Формальні параметри застосовують лише в тілі процедури, ім'я процедури і формальні параметри – ідентифікатори. Процедуру n обчислює оператор виклику n(a1, ..., at), де n – ім'я процедури, a1, ..., at – фактичні параметри (аргументи). Кількість аргументів має дорівнювати кількості формальних параметрів. Аргументами в цій мові слугують тільки змінні, тобто параметри передаються за посиланням (як змінні).

Програма, яка вводить ціле число n, обчислює й виводить всі менші від n прості числа, в процедурній мові має вигляд:

```
{ int n, i, s;
  proc smp(i1, s1)
    { int j; j := 2; s1 := 1;
      while (i1 – j) {
          if (1 – i1 % j) { s1 := 0; j := i1}
          j := j+1
      }
    }
  read n; i := 2;
  while (n – i) {
      smp (i, s);
      if (s) write i;
      i := i+1
  }
}
```

Тут є п'ять блоків. Першим є програма, в ньому описано локальні змінні n, i, s та процедуру smp; другим – тіло процедури smp з описом локальної змінної j; інші блоки містять лише списки операторів – третій і четвертий в операторах циклу і умовного в тілі процедури, п'ятий в операторі

циклу програми. Параметрами процедури smp є i1, s1, і після її обчислення s1 дорівнює 1, якщо i1 – просте число, або 0 – у протилежному разі.

Опис нових даних у різних місцях програми пов'язаний із правилами області дії, які встановлюють співвідношення місця їх опису, і частиною тексту програми, де вони відомі та доступні для використання. Областю дії імені локального даного, описаного в блоці, є текст блоку, зокрема локальні процедури, крім вкладених блоків, де це ім'я визначають повторно. Наприклад, у фрагменті програми

```
1    { int v; ...
2      ... v ...
3      { int v; .
4        ... v ... }
5     ... v ...}
```

із двома блоками і двома описами змінної v областю дії першої з них є рядки 1, 2, 5, а другої – рядки 3, 4.

Із блоком пов'язаний механізм керування пам'яттю (розподілу пам'яті), у якій зберігаються значення змінних. Найчастіше використовують автоматичний розподіл із локальними автоматичними змінними. Пам'ять їм виділяють кожного разу на початку виконання блоку, в якому вони описані, і звільняють (вона стає недоступною) після закінчення виконання блоку. Наприклад, за виконання фрагмента програми

```
1    { int a, b, c; ....
2      { int a;  ... a ...};
3      ... a ... b ... c...
4      { int b;  ... b ...};
5      ... a ... b ... c...
6    }
```

розподіл пам'яті виконують так. На початку виконання блоку 1 виділяють ділянки пам'яті la, lb, lc описаним у ньому локальним змінним a, b, c; при вході в блок 2 локальній змінній a надають ділянку la', яку звільняють після його закінчення. У блоці 4 змінна b дістає пам'ять lb', яка існує, доки працює блок. За виконання операторів із рядків 1, 3, 5 виділеними є ділянки la, lb, lc; із рядка 2 – la, lb, lc, la'; з рядка 4 – la, lb, lc, lb'. Отже, одне ім'я може посилатися на різні значення, коли йому надано декілька ділянок пам'яті. Так, у рядку 2 із a пов'язані la і la', у рядку 4 із b – lb і lb'. Зауважимо, що при виконанні блоку 2 значення змінної a, описаної в 1, стає недоступним, а при виконанні блоку 4 буде сховане значення b з блоку 1.

За наявності в мові процедур і параметрів, які передають за посиланням, можуть виникати ситуації, коли різні імена зв'язані з одним значенням, тобто їм виділено спільну ділянку пам'яті. Наприклад, при виконанні виклику процедури p в програмі

```
1    { int a;
2      proc p(x, y)
3          x :=  a + y
4      p(a, a)
5    }
```

в середині процедури (рядок 3) імена x, y, а посилаються на один об'єкт a, і зміна значення якого-небудь із них приводить до зміни всіх інших.

Конкретний і абстрактний синтаксис

Основа мови програмування – конкретний синтаксис, який виділяє синтаксично правильні послідовності символів алфавіту мови. Часто для задання конкретного синтаксису використовують розширену форму Бекуса–Наура [1]. Конкретний синтаксис – це множина синтаксичних правил виду

name = defin ;

де name – ім'я синтаксичної конструкції (ідентифікатор), a defin – визначення, яке описує name. Визначення – це послідовність символів алфавіту, імен синтаксичних конструкцій і метасимволів ',', '|', '[', ']', '{', '}'. Застосування таке: ',' з'єднує конструкції, '|' розділяє допустимі варіанти запису конструкції, дужки [ ] охоплюють послідовність символів, яка або входить в конструкцію або ні, а дужки { } охоплюють послідовність символів, яка або входить у конструкцію декілька разів або не входить жодного.

Конкретний синтаксис процедурної мови містить конструкції – програми *program*, оператора *stmt*, *assignSt, callSt, definS,* означення процедури *defPrS*, *procedS,* виразу *expr*, доданку *term* і множника *factor*.

program= *stmt ;*
stmt = 'while' , '(' , *expr* , ')' , *stmt* ;
| 'if' , '(' , *expr* , ')' , *stmt* ;
| 'read' , *identifier* ;
| 'write' , *expr* ;
| *identifier* , (*assignSt* | *callSt*) ;
| '{' , [*definS*] , { *defPrS* } , *stmt* , {';' , *stmt* } , '}' ;
assignSt = ':=' , *expr* ;
callSt = '(' , *identifier* , { ',' , *identifier* } , ')' ;
definS = 'int' , *identifier* , { ',' , *identifier* } , ';' ;
defPrS = 'proc' , *identifier* , *procedS* ;
procedS = '(' , [*identifier* , { ',' , *identifier* }] , ')' , *stmt* ;
expr = *term* , {*addOp* , *term*} ;
term = *factor* , {*mulOp* , *factor*} ;
factor = *decimal* | '(' , *expr* , ')' | *identifier* ;

Ці означення використовують базові (прості) конструкції – операції додавання *addOp* і множення *mulOp*, змінної *identifier* та цілого *decimal*.

```
addOp     = '+' | '_' ;
mulOp     = '*' | '/' | '%' ;
identifier =  letter ,  { ( digit | letter ) } ;
              крім 'int' 'if' 'while' 'read' 'write' 'proc'
decimal  =  digit , {  digit } ;
letter     = 'A' |  ... | 'Z' | 'a' | ... | 'z' ;
digit      = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' |
'9' ;
```

Для опису семантики мови перевагу віддають абстрактному синтаксису, який у реальних мовах програмування коротший і більш наочний, ніж конкретний. Зв'язок між об'єктами абстрактного синтаксису і конкретним синтаксисом програми в компіляторах встановлює фаза синтаксичного аналізу.

Абстрактний синтаксис визначаємо типами: програми Program, процедури Proc, оператора Stmt, операції Op та виразу Expr.

```
data Op   =  Plus | Minus | Times | Div | Mod
                deriving (Show, Eq)
data Expr  =  Var String
            |  Const  Integer
            |  BinOp Op Expr Expr
               deriving (Show, Eq)
data Stmt  =  Assign  String  Expr
            |  Read String
            |  Write  Expr
            |  If   Expr  Stmt
            |  While   Expr Stmt
            |  Call String [String]
            |  Block [String] [(String Proc)] [Stmt]
               deriving (Show, Eq)
type Program = Stmt
```

Синтаксичний аналіз

Для реалізації синтаксичного аналізу в Haskell [2] можна скористатися бібліотекою parser. Синтаксичний аналізатор p, що розпізнає значення типу a на початку рядка типу String, має тип Parser a, який є екземпляром класу типів Monad. У разі успіху залишок рядка, що аналізується, передають наступному аналізатору в обчислювальному контексті, який підтримує монада Parser.

На першому кроці бібліотеки Text. ParserCombinators.Parsec.Language і Text. ParserCombinators.Parsec.Token будуємо аналізатори, які розпізнають лексичні одиниці мови.

```
import Text.ParserCombinators.Parsec.Language
 import qualified Text.ParserCombinators.Parsec.
Token as P
import Text.ParserCombinators.Parsec
 import qualified Text.ParserCombinators.Parsec.
Expr as E

lexer :: P.TokenParser ()
lexer =  P.makeTokenParser (
      emptyDef { identStart = letter
          , identLetter = alphaNum
          , reservedNames = [«int»,»read»,»write»
,»if»,while»,»proc» ]
     , reservedOpNames = [«*»,»/»,»%»,»+»,»-»]}  )
```

Функція emptyDef будує запис типу LanguageDef, який описує лексичну основу мови. Для цього функції emptyDef передають інформацію про ідентифікатори – починаються з букви (identStart = letter) і містять лише букви і цифри (identLetter = alphaNum); службові слова (reservedNames) та операції (reservedOpNames). Функція makeTokenParser, використовуючи створену лексичну основу, будує запис типу TokenParser, який містить 29 лексичних аналізаторів, які можна використовувати при побудові інших аналізаторів. Кожний із них вибирається відповідним селектором.

На наступному кроці будуємо аналізатори виразів. Кожний вираз у мові будують із простих виразів-множників *factor* і бінарних операцій згідно з їхніми пріоритетами та асоціативністю. Використовуючи лексичні аналізатори P.parens lexer p (розпізнає конструкцію p у дужках), P.decimal lexer (розпізнає додатне ціле число в десятковій системі й повертає його значення) і P.identifier lexer (розпізнає правильний ідентифікатор, але не допускає зарезервовані слова), будуємо аналізатор factorS, що розпізнає простий вираз-множник *factor*, який є – вираз у дужках, число або змінна (ідентифікатор).

```
factorS :: Parser Expr
factorS  =  P.parens lexer expr
        <|> (P.decimal lexer >>= return . Const)
        <|> (P.identifier lexer >>= return . Var)
```

Бібліотека Text.ParserCombinators.Parsec.Expr надає можливість будувати таблицю table синтаксичних аналізаторів усіх бінарних операцій, які використовують у виразах мови. Кожний елемент таблиці – це список аналізаторів операції одного пріоритету. Для кожного аналізатора потрібно вказати асоціативність його операції (всі лівоасоціативні E.AssocLeft). Елементи таблиці впорядковано згідно зі зменшенням пріоритету.

Функція buildExpressionParser будує синтаксичний аналізатор виразів exprS, який використовує аналізатор простих виразів factorS і аналізатори таблиці table.

```
table :: E.OperatorTable Char () Expr
 table  = [ [op «*» Times E.AssocLeft, op «/» Div
E.AssocLeft
                        , op «%» Mod E.AssocLeft]
          , [op «+» Plus E.AssocLeft, op «-» Minus
E.AssocLeft]
```

```
        ]
    where  op s f assoc = E.Infix (binOp s f) assoc
           binOp s f = P.reservedOp lexer s >> return
(Syntax.BinOp f)
  exprS :: Parser Expr
  exprS  = E.buildExpressionParser table factorS
```

На заключному кроці, використовуючи аналізатор exprS і лексичні аналізатори з запису lexer, будуємо аналізатори операторів (stmtS, assignStS, callStS, definS), об'яви процедури (defPrS, procedS) і програми (programL).

```
 stmtS :: Parser Stmt
  stmtS = (P.reserved lexer «while» >> While <$>
P.parens lexer exprS <*> stmtS)
     <|> (P.reserved lexer «if» >> If <$> P.parens lexer
exprS <*> stmtS)
         <|> (P.reserved lexer «read» >> Read <$>
P.identifier lexer)
     <|> (P.reserved lexer «write»>> Write <$> exprS)
      <|> (P.identifier lexer >>= (\v -> assignStS v <|>
callStS v))
      <|> (P.braces lexer (Block <$> (option [] definS)
                          <*> (many defPrS) <*>
P.semiSep lexer stmt))

 assignStS :: String -> Parser Stmt
  assignStS var = symbol «:=» >> Assign var <$>
exprS
 callStS :: String -> Parser Stmt
 callStS var = P.parens lexer  (Call var <$>
                    (option [] (P.commaSep1 lexer
(P.identifier lexer))))
 definS :: Parser [String]
 definS = P.reserved lexer «int»
          *> P.commaSep1 lexer (P.identifier lexer)
<* P.symbol lexer «;»
 defPrS :: Parser (String,Proc)
   defPrS = P.reserved lexer «proc» >> (,) <$>
(P.identifier lexer) <*> procedS
 procedS :: Parser Proc
 procedS = (,) <$>  P.parens lexer
                    (option [] (P.commaSep1 lexer
(P.identifier lexer)))  <*> stmtS
 programL :: Parser Program
 programL = P.whiteSpace lexer *> stmtS <* eof
```

Контекстні умови

Прикрим є те, що повний опис деяких складних конструкцій процедурної мови контекстно вільним синтаксисом (конкретним або абстрактним) дати не можна. В такому разі використовують контекстно-залежні правила, що описують потрібні додаткові контекстні умови.

Контекстні умови – це предикати, які накладають на об'єкти, визначені правилами абстрактного синтаксису, додаткові контекстно залежні обмеження. В подальшому, коли задають семантику, вважають, що використовують лише об'єкти, які задовольняють ці обмеження. Більшість контекстних умов пов'язана з правильним уживанням даних у програмі. Наприклад, їх задовольняє вираз a-b+2, якщо він розміщений у програмі там, де a і b – відомі імена змінних або параметрів. Оператор виклику процедури p(b) задовольняє контекстні умови, якщо в навколишньому середовищі b буде змінною або параметром, а p – іменем процедури з одним параметром.

Для зберігання й передавання інформації про дані програми (Static) застосовують «статичне» середовище (EnvStatic).

```
 data Static  = VarSt | ProcSt Int deriving (Show, Eq)
 type EnvStat = [(String,Static)]
```

Використаному в контекстних умовах середовищу env відповідає список усіх відомих у цій точці програми імен даних. Ідентифікатор id – ім'я змінної або параметра, якщо в списку є пара (id, VarSt), або ім'я процедури з k≥ 0 параметрами, якщо в списку є пара (id, ProcSt k).

Контекстні умови задаємо для об'єктів: Program (iswfProgram), Proc (iswfProc), Stat (iswfStat) і Expr (iswfExpr). Вони використовують допоміжні предикати: distinct is – перевіряє, що список is не має дублікатів, та iswfData id st env – перевіряє, що ідентифікатор id відомий як змінна (st=VarSt) чи як процедура з k параметрами (st=ProcSt k).

```
 distinct :: Eq a => [a] -> Bool
 distinct []     = True
 distinct (v:vs) = notElem v vs && distinct vs
 iswfData :: String -> Static -> EnvStat -> Bool
 iswfData st sd env = maybe False (==sd) (lookup st
env)
 iswfProgram :: Program -> Bool
 iswfProgram pr = iswfStmt pr []
 iswfProc :: Proc -> EnvStat -> Bool
 iswfProc (px,st) env = let psx = map (\v -> (v,VarSt))
px
                  in distinct (map fst psx) && iswfStmt
st (psx++env)
 iswfStmt :: Stmt -> EnvStat -> Bool
  iswfStmt (Assign var e) env = iswfData var VarSt
env && iswfExpr e env
 iswfStmt (If e s) env            = iswfExpr e env &&
iswfStmt s env
 iswfStmt (While e s) env         = iswfExpr e env &&
iswfStmt s env
 iswfStmt (Call p vs) env         = iswfData p (ProcSt
(length vs)) env
                                  && all (\v -> iswfData
v VarSt env) vs
 iswfStmt (Block vs ps sts) env =
     let pts = map (\p ->(fst p, ProcSt $ length $ fst $
snd p)) ps
         new = map (\v -> (v,VarSt)) vs ++ pts
     in  distinct (map fst new)
```

```
        && all (((flip iswfProc (new++env) . snd) )) ps
          && all (flip iswfStmt (new++env)) sts
  iswfStmt (Read var) env      = iswfData var VarSt env
  iswfStmt (Write e) env       = iswfExpr e env
   iswfExpr :: Expr -> EnvStat -> Bool
   iswfExpr (Var s) env        = iswfData s VarSt env
   iswfExpr (Const _) _        = True
    iswfExpr (BinOp _ e1 e2) env = iswfExpr e1 env
&& iswfExpr e2 env
```

Денотати

Завданням семантики мови програмування є пов'язати з конструкціями мови певні математичні об'єкти (списки, таблиці, функції), їх називають денотатами, які задають «значення» (семантику) конструкцій. Фактично денотатами є класи об'єктів, у термінах яких пояснюють усі конструкції мови (задають їхню семантику). Семантика мови – набір семантичних функцій, які відображають синтаксичні конструкції мови, зокрема й програму, у відповідні денотати.

```
type Work = ([Integer], [Maybe Integer], [Integer])
```

Для опису семантики процедурної мови використовують стан – дане типу Work. Це кортеж із трьох елементів (inp, stg, out), який моделює три набори значень, з якими працює програма: inp – список цілих [Integer] задає вхідні дані (файл введення), stg – список значень [Maybe Integer] зберігає поточні значення всіх змінних програми (пам'ять), out – список цілих [Integer] задає результуючі дані (файл виведення). Зауважимо, що значення кожної змінної програми – це елемент списку stg. Якщо поточне значення деякої змінної є ціле v, то в списку stg її поточне значення відображають як Just v, а якщо змінній ще не було присвоєно жодного значення, то поточне значення відображають як Nothing.

Для роботи зі станом семантичні функції використовують функції, які управляють пам'яттю – другим компонентом робочого стану allocate, getBase і free; працюють зі значенням змінних getValue і updValue; працюють із файлом виведення writeValue та файлом введення readInput, і dropInput.

```
allocate :: Int -> Work -> Work
allocate k = \(inp,stg,out) ->
                  let beg = [Nothing | _ <- [1..k]]
                  in (inp, beg++stg,out)
getBase :: Work -> Int
getBase = \(_,stg,_) -> length stg
free :: Int -> Work -> Work
free k = \(inp, stg,out) -> (inp, drop k stg,out)
getValue:: Int -> Work -> Integer
getValue k = \(_,stg,_) ->  let p = length stg – k
                            in case stg!!p of
                                 Just v  -> v
                                 Nothing -> error «valueNothing»
updValue :: Int ->  Integer -> Work -> Work
updValue k v = \(inp,stg,out) -> let p = length stg – k
                   (beg,end) = splitAt p stg
                      stg1 = beg ++ [Just v] ++ (tail end)
                                 in  (inp,stg1,out)
writeValue :: Integer -> Work -> Work
writeValue v = \(inp, stg, out) -> (inp, stg, out ++ [v])
readInput :: Work -> Integer
  readInput = \(inp,_,_) -> if null inp then error
«readInput» else head inp
dropInput :: Work -> Work
dropInput = \(inp,stg,out) -> (tail inp,stg,out)
```

Управління динамічною пам'яттю в робочому стані (inp,stg,out) реалізують другим компонентом stg. Функції allocate k (inp,stg,out) і free k (inp,stg,out) працюють зі списком stg як зі стеком: перша – на початку виконання блоку, який вводить локальні змінні v1, ..., vk, додає в список stg k елементів для розміщення значень k локальних змінних блоку, а друга після закінчення виконання блоку вилучає їх. Функція getBase (inp,stg,out) повертає довжину списку stg. Якщо список stg, перед виконанням блоку, який вводить локальні змінні v1, ..., vk, має b (результат функції getBase) елементів stg=[a1, ..., ab], то після виконання функції allocate список stg буде мати (k+b) елементів stg = [vk, ..., v1, f1, ..., ab]. vi – позначає елемент списку stg, в якому буде міститися поточне значення локальної змінної vi в процесі виконання блоку. Коли необхідно в процесі виконання блоку отримати доступ до змінної vi з адресом k, досить взяти елемент списку stg з індексом p = length stg – k.

Функція getValue k (inp,stg,out) знаходить поточне значення змінної адресою k. Якщо це значення Just v, то повертає v, а якщо Nothing, то генерує помилку "valueNothing". Функція updValue k v (inp,stg,out) реалізує зміну поточного значення змінної з адресою k. Значення елементу списку stg з індексом p = length stg – k покладають Just v.

Функція writeValue v (inp,stg,out) змінює стан, додаючи в кінець списку out ціле значення v.

Функція readInput (inp,stg,out) повертає перше значення зі списку inp, якщо список порожній, то генерує помилку "readInput". Функція dropInput (inp,stg,out) змінює стан, вилучаючи перший елемент списку inp. Використовують відразу після функції readInput.

Семантичні функції

Наявність блоків і процедур приводить до того, що в конкретних програмах одне ім'я може посилатися на різні значення (опис змін-

ної у внутрішньому блоці), а різні імена – на одне значення (параметри, які передають як змінні). Для пов'язування імен даних з їхніми денотатами («значеннями») вводять середовище – значення типу Env. Кожне середовище env – це список пар (id, d), де id – ім'я об'єкта (змінна або процедура) і d – денотат об'єкта. Денотат змінної є Location k, k – адреса, де зберігають її значення в другій компоненті стану. Денотат процедури є Procedure f, f – це функція зміни стану, яка залежить від денотатів фактичних параметрів.

```
data Denote = Location Int
            | Procedure ([Denote] -> Work -> Work)
type Env = [(String, Denote)]
```

У семантичних функціях використовують такі допоміжні функції:

```
getDenote :: String -> Env -> Denote
getDenote s env = fromJust $ lookup s env
getProcedure:: String -> Env -> ([Denote] -> Work
-> Work)
getProcedure s env = case fromJust $ lookup s env of
          Location _ -> error «getProcedure»
          Procedure f -> f
getLocation :: String -> Env -> Int
getLocation s env = case fromJust $ lookup s env of
          Location k -> k
          Procedure _ -> error «getLocation»
applyBo :: Op -> Integer -> Integer -> Integer
applyBo Plus v1 v2  = v1 + v2
applyBo Minus v1 v2 = v1 – v2
applyBo Times v1 v2 = v1 * v2
applyBo Div v1 v2   = if v2 /= 0 then div v1 v2 else
error «DivOnZero»
applyBo Mod v1 v2   = if v2 /= 0 then mod v1 v2 else
error «ModOnZero»
extEnv :: [String] -> [(String,Proc)] -> Int -> Env ->
Env
extEnv vs ps b = \env ->
          let nenv1 = (zip vs [Location (b+i) | i<-
[1..]]) ++ env
              nenv  = (map (\(n,pr) -> (n, eProc pr
nenv)) ps) ++ nenv1
          in nenv
```

Для доступу до денотатів об'єктів програми або їх компонентів використовують функції getDenote id env, що повертає денотат об'єкта з іменем id, getLocation id env – повертає адрес змінної id, getProc p env – повертає функцію зміни стану, залежну від списку денотатів змінних-аргументів. При обчисленні виразу використовують функцію applyBop op v1 v2, яка обчислює результат застосування бінарної операції op до цілих значень v1 і v2. У разі обчислення операції Div або Mod, якщо другий операнд рівний нулю, то генерують помилку «DivOnZero» або «ModOnZero». Функція extEnv vs ps b env за середовищем env, в якому потрібно виконати блок із локальними змінними vs і процедурами ps, і розміром b списку, який моделює пам'ять, будує нове розширене середовище.

Денотатами виразів Expr і операторів Stmt процедурної мови є відповідно функції зміни стану типу Work -> Integer і Work -> Work, а денотатом програми Program – функція типу [Integer] -> [Integer]. Ці денотати будують семантичні функції eExpr, iStmt і iProgram відповідно. Функція eProc (ps,st) env будує денотат процедури з формальними параметрами ps і тілом оператор st. Це об'єкт виду Procedure f, де f функція зміни стану – денотат оператору st, який будує семантична функція iStmt в розширеному середовищі. Більшість семантичних функцій для доступу до денотатів змінних і процедур використовують середовище як додатковий параметр.

```
eExpr :: Expr -> Env -> Work -> Integer
eExpr (Var s) env          = \w -> getDValue s env w
eExpr (Const v) _          = \_ -> v
eExpr (BinOp op e1 e2) env = \w ->
                    applyBo op (eExpr e1 env w)
(eExpr e2 env w)
eProc :: Proc -> Env -> Denote
eProc (ps,st) env = Procedure $ \ds -> iStmt st ((zip
ps ds)++env)
iStmt :: Stmt -> Env -> Work -> Work
iStmt (Assign var e) env    = \w -> updDValue var
(eExpr e env w) env w
iStmt (If e s) env          = \w -> if eExpr e env w > 0
                                    then iStmt
s env w else w
iStmt wh@(While e s) env = \w -> if eExpr e env w > 0
                                    then iStmt wh
env (iStmt s env w) else w
iStmt (Call p vs) env    = \w -> let f  = getProc p env
                                     ds = map (flip
getDenote env) vs
                                 in f ds w
iStmt (Block vs nps sts) env  = \w ->
                              let k    = length vs
                                  w1   = allocate k w
                                  nenv = extEnv vs
nps (getBase w) env
                                  w2 = foldl (\wr
s -> iStmt s nenv wr) w1 sts
                              in free k w2
iStmt (Read var) env      = \w -> let v = readInput w
                                  w1 = updDValue
var v env w
                              in dropInput w1
iStmt (Write e) env         = \w -> writeValue (eExpr
e env w) w
iProgram :: Program -> [Integer] -> [Integer]
iProgram prog ix = let w = (ix, [],[])
                       (_,_,ox) = iStmt prog [] w
                   in ox
```

Реалізація

Специфікація процедурної мови програмування, що використовує мову Haskell [2], містить описи синтаксису й семантики мови. Синтаксис містить: конкретний синтаксис, яким описують синтаксичні правила в розширеній БНФ; абстрактний синтаксис, який містить типи – Op (бінарний оператор), Expr (вираз), Stmt (оператор), Proc (процедура) і Program (програма); контекстні умови, які накладають на об'єкти, визначені правилами абстрактного синтаксису, додаткові контекстно залежні обмеження. Семантика містить: основу денотатів мови – тип Work, який задає стан програми процедурної мови, та базові функції, які працюють із ним; денотати об'єктів мови (змінні та процедури) – тип Denote та синтаксичних конструкцій мови (Expr, Stmt, Program) – функції зміни стану Work -> Integer, Work -> Work і [Integer] -> [Integer]; семантичні функції, які будують денотати синтаксичних конструкцій і об'єктів – eExpr, eProc, iStmt, iProgram.

Зауважимо, що всі функції базові, денотати і семантичні – чисті функції. Використавши чистою функцією parsePLL, яка реалізує синтаксичний аналіз процедурної мови, будуємо інтерпретатор interpret (тобто реалізацію) процедурної мови. Якщо програма (рядок s) синтаксично правильна, то інтерпретатор, застосувавши до результату синтаксичного аналізу pr семантичну функцію iProgram, будує денотат програми, який потім застосовує (інтерпретує, виконує) до вхідних даних ix – iProgram pr ix. interpret – чиста функція, але якщо на кроці синтаксичного аналізу, перевірки контекстних умов або при застосуванні денотату виникає помилка, то обчислення переривається, генеруючи відповідне повідомлення (функція error).

```
parsePLL :: String -> Program
parsePLL s = case parse program «» s of
                  Left _  -> error «Syntax»
                  Right e -> e
interpret :: String -> [Integer] -> [Integer]
interpret st ix = let pr = parsePLL st
                      wf = iswfProgram pr
                  in if wf then iProgram pr ix
                           else error «Contex»
```

За посиланням https://github.com/ProtsenkoVS/lngProc.git можна завантажити проєкт lngProc, який містить опис процедурної мови програмування, зокрема різні варіанти синтаксичного аналізу і семантичних функцій. Проєкт забезпечує можливість роботи з інтерпретатором у режимі REPL, введення програм з файлу, оброблення переривань і зручний вибір різних варіантів роботи.

Список літератури

1. ISO/IEC 14977:1996 Information technology–Syntatic metalanguage – Extended BNF.
2. Kurt W. Get Programming with Haskell / W. Kurt. – Manning Publications, 2018. – 296 p.

References

ISO/IEC 14977:1996 Information technology–Syntatic metalanguage – Extended BNF.
Kurt, W. (2018). *Get Programming with Haskell.* Manning Publications.

V. Protsenko

SPECIFICATION OF THE PROCEDURAL PROGRAMMING LANGUAGE

A simple procedural programming language is considered, each program of which can input integer values, process them and output new integer values as result. A program is a block with description of local integer variables and procedures and a list of statements. The language has data processing statements: assignment, input, output, conditional, loop, procedure call and block. Main purpose of the block is to enter local data (integer variables and procedures) that are used in the body of the block – a list of operators. The scope of the name of the local data described in the block is the text of the block except for nested blocks, where this name is redefined. A mechanism of automatic memory allocation for variables entered in the block is also associated with the block. Memory for local variables is allocated when entering a block, and freed when exiting a block. A block containing only a list of statements is valid. The procedure has a name, list of formal parameters, and a body – a statement (most often a block). Formal parameters are applied

only in its body. A procedure is calculated by the procedure call statement, whose actual parameters are only variables. Parameters are passed by reference (pass-by-reference).

A formal specification of a programming language is a description of its syntax and semantics. A concrete syntax, finite set of rules, singles out syntactically correct sequences of symbols of the alphabet of language. To describe the semantics of a language, as a rule, abstract syntax is used, adding contextual conditions to it. The task of semantics is to introduce the denotations ("meanings") of the basic constructions of language and semantic functions that build the denotations of complex syntactic constructions based on the denotations of their components, including the program.

The article provides a specification of a procedural programming language that uses the extended Backus-Naur form to describe a concrete syntax, and the tools of the functional language Haskell to describe other parts. Abstract syntax is defined by the types Program, Proc, Stmt, Expr and Op. Additional contextual conditions are predicates that use information about program data. Most of the context conditions are related to the correct use of data in the program. The leading predicate that checks the context conditions of the program pr is iswfProgram pr.

The language denotations are based on the Work type. The value of this type – a tuple (inp, stg, out) models the environment in which the language program is executed: inp - input data, stg – memory containing variable values, out – resulting data. The semantics of main constructions procedure, statement and expression are functions of the type Work -> Work or Work -> Integer. The semantics of the program is a function of the type [Integer] -> [Integer]. Semantic functions build these denotations according to syntactic constructions, which are described by abstract syntax – Proc, Stmt, Expr, Program types. The semantics of the program (Program) pr is built by function iProgram pr.

All functions: contextual conditions, denotations and semantic functions are pure functions. Using Haskell tools, a function called parsePLL is built, which connects concrete and abstract syntax. It is shown how by combining the functions parsePLL, iswfProgram and iProgram you can get a procedural language – a pure function with the name interpret.

Keywords: programming language, program, procedure, statement, expression, syntax, denotation, semantic function, Haskell, parse, interpreter.

Матеріал надійшов 15.08.2022