УДК 519.682.1
DOI: 10.18523/2617-3808.2025.8.149-157

Проценко В. С.

ДЕНОТАЦІЙНА СЕМАНТИКА ОДНОВИМІРНИХ МАСИВІВ

Розглянуто імперативну мову програмування, об'єктами якої є цілі змінні — скалярні та одновимірні масиви. Оператори мови — присвоєння, введення, виведення, умовний, циклу і блок. Призначення боку — введення локальних цілих змінних. Одновимірний масив a[k], де k>0 — ціле додатне число (розмірність масиву). Робота з масивами здійснюється поелементно. Доступ до окремого елементу масиву здійснює операція індексування a[e], e — цілий вираз. Значення e — число від 0 до k-1. Наведено повну формальну специфікацію мови. На основі специфікації побудовано інтерпретатор мови програмування.

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

Мова з одновимірними масивами

Розглянуто імперативну мову програмування, що має цілі скалярні дані і одновимірні масиви. Ця мова є розширенням мови зміни станів [1] одновимірними масивами. Кожна змінна a в цій мові — або ціла скалярна змінна a, або цілий одновимірний масив a[k], де k>0 — ціле додатне число (розмірність масиву). Такий масив має k елементів, які розміщуються послідовно і нумеруються як a[0], a[1], ..., a[k-1]. Робота з масивами здійснюється тільки поелементно.

Кожна програма мови може вводити цілі значення, обробляти їх і виводити цілі значення. Програма — це окремий оператор, як правило, блок із описом локальних цілих змінних (скалярних або масивів) та списком операторів — тілом блоку. Головну обробку 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. Усі вирази e в операторах обчислюють скалярне ціле значення. Якщо a[k] — цілий одновимірний масив, то доступ до окремого елементу масиву здійснює операція індексування — запис виду a[e]. Значення iv цілого виразу e — повинно бути ціле число від 0 до k-1. В операторах умовному і циклу значення iv>0 цілого виразу e еквівалентно логічному значенню True.

Далі наведено приклад програми в цій мові. Програма вводить масив a з 10 елементів, в окремому блоці (з локальними змінними is і c) впорядковує його, використовуючи «бульбашкове» сортування, і виводить впорядкований масив.

```
{ int i, a[10];
  i := 0; while (10 - i) {read a[i]; i:= i+1 };
  {  int is, c; is := 1;
     while (is) { is := 0: i := 0;
          while (9 – i) {
        if (a[i] – a[i+1]) {
           is := 1; c := a[i+1]; a[i+1] := a[i]; a[i] := c };
        i := i+1} }
 };
 i := 0; while (10-i) {write a[i]; i := i+1}
}
```

Опис мови програмування охоплює синтаксис і семантику. Синтаксис мови програмування — це опис усіх конструкцій, що утворюють елементи мови. Семантика мови вказує «значення» синтаксичних конструкцій. Для опису семантики використовується денотаційна семантика. Спочатку фіксуються денотати (як правило, математичні об'єкти) найпростіших синтаксичних об'єктів. Потім із кожною складеною синтаксичною конструкцією пов'язується семантична функція, яка за денотатами компонентів конструкції обчислює її значення — денотат. Оскільки програма є конкретною

синтаксичною конструкцією, то її денотат можна визначити, застосувавши відповідну семантичну функцію. Зауважимо, що сама програма при обчисленні її денотата не виконується.

Конкретний і абстрактний синтаксис

Конкретний синтаксис, що описується за допомогою БНФ [2], можна розділити на три частини: лексика, вирази і оператори.

У лексиці описуються все лексичні конструкції, які використовує мова.

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

Обчислення значень виконують вирази.

var = *identifier* , [*'['* , *expr* , *']'*] ;
expr = *term* , {*addOp* , *term*} ;
term = *factor* , {*mulOp* , *factor*} ;
factor = *decimal* | '(' , *expr* , ')' | *var* ;

Послідовність дій описують оператори

program = *stmt ;*
stmt = 'while' , '(' , *expr* , ')' , *stmt* ;
| 'if' , '(' , *expr* , ')' , *stmt* ;
| 'read' , *identifier* ;
| 'write' , *expr* ;
| *var* , ':=' , *expr* ;
| '{' , [*defin*] , { *defPrS* } , *stmt* , {';' , *stmt* } , '}' ;
defin = 'int' , *defvar* , { ',' , *defvar* } , '; ' ;
defvar = *identifier* , [*'['* , *decimal* , *']'*] ;

callSt = '(' , *identifier* , { ',' , *identifier* } , ')' ;
definS = 'int' , *identifier* , { ',' , *identifier* } , '; ' ;
defPrS = 'proc' , *identifier* , *procedS* ;
procedS = '(' , [*identifier* , { ',' , *identifier* }] , ')' , *stmt* ;

Для опису семантики мови віддають перевагу абстрактному синтаксису. Зв'язок між об'єктами конкретного і абстрактного синтаксису виконує синтаксичний аналіз.

Абстрактний синтаксис визначається типами: бінарної операції Op, змінної Var, виразу Expr, оголошення змінної VarDef, оператору Stmt і програми Program.

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

Синтаксичний аналіз

Для реалізації синтаксичного аналізу використана бібліотека Parsec. Синтаксичний аналіз у цій бібліотеці виконують спеціальні функції — аналізатори. Синтаксичний аналізатор p, що розпізнає значення типу a на початку рядка типу String, має тип Parser a. У випадку успіху залишок рядка, що аналізується, передається наступному аналізатору.

На першому кроці будуються аналізатори, що розпізнають лексичні одиниці мови. Мова з одновимірними масивами дозволяє вживати проміжки між довільними лексичними одиницями. Синтаксичний аналізатор lexem p за аналізатором p будує аналізатор, який розпізнає конструкцію p і всі проміжки за нею. identif — аналізатор, що розпізнає послідовність букв і десяткових цифр, котра починається з букви. Синтаксичний аналізатор identifier — розпізнає ідентифікатори, що не збігаються із зарезервованими словами, відповідно аналізатор reserved st — розпізнає зарезервоване слово st.

```
identif :: Parser String
identif = do {c <- letter; cs <- many alphaNum; return (c:cs)}
lexem :: Parser a -> Parser a
lexem p = do {a <- p; spaces; return a}
identifier :: Parser String
identifier = try (do {nm <- lexem identif;
                 if (any(nm==) [«int»,»read»,»write»,»if»,»while»])
                  then unexpected («reserved word « ++ show nm) else return nm} )
reserved :: String -> Parser ()
reserved st = try( lexem (do {_ <- string st; notFollowedBy alphaNum}))
symbol :: String ->  Parser ()
symbol st = lexem (do {_ <- string st; return ()})
oper  :: String -> Op -> Parser (Expr -> Expr -> Expr)
oper str bop = do {symbol str; return (BinOp bop)}
mulOp, addOp :: Parser (Expr -> Expr -> Expr)
mulOp = (oper «*» Times) <|> (oper «/» Div) <|> (oper «%» Mod)
addOp = (oper «+» Plus) <|> (oper «-» Minus)
```

На наступному кроці будуємо аналізатори виразів. Головні з них: decimal — розпізнає число, var — розпізнає змінну, можливо з індексом, factor, term, expr — розпізнають різні частини виразу.

```
parens, brackets :: Parser a -> Parser a
parens p = do {symbol «(«; e <- p; symbol «)»; return e}
brackets p = do {symbol «[«; e <- p; symbol «]»; return e}
decimal :: Parser Integer
decimal = do {ds <- many1 digit; return (read ds)}
dimension :: Parser Int
dimension = do {ds <- many1 digit; return (read ds)}
var  :: Parser Var
var  = do {v  <- identifier;
      me <- option Nothing (do {e<-brackets expr; return (Just e)}) ;
      return (v,me) }
factor, term, expr :: Parser Expr
factor  = parens expr  <|> do {v <- lexem decimal; return (Const v)}
     <|> do {v <- var; return (VarOp v)} <?> «factor»
term = chainl1 factor mulOp
expr = chainl1 term addOp
```

На заключному кроці будуються аналізатори операторів, оголошень даних і програми. Ведуча функція parseProgram s запускає основний аналізатор program, що розпізнає в рядку s програму, і аналізує результат його роботи. Якщо рядок s містить синтаксично правильну програму, то функція повертає її представлення в абстрактному синтаксисі — дане типу Program. У випадку синтаксичної помилки процес обчислення переривається і генерується інформація про помилку.

```
braces :: Parser a -> Parser a
braces p = do {_ <- symbol «{«; e <- p; _ <- symbol «}»; return e}

semiSep1, commaSep1 :: Parser a -> Parser [a]
semiSep1 p = sepBy p (symbol «;»)
commaSep1 p = sepBy p (symbol «,»)
stmt :: Parser Stmt
stmt = do {reserved «while»; e <- parens expr; s <- stmt; return (While e s)}
   <|> do {reserved «if»; e <- parens expr; s <- stmt; return (If e s)}
   <|> do {reserved «read»; v <- var; return (Read v)}
   <|> do {reserved «write»; e <- expr; return (Write e)}
   <|> do {v <- var; symbol «:=»; e <- expr; return (Assign v e)}
   <|> braces (do {dll <- option [] defin;
                   sl <- semiSep1 stmt; return (Block dll sl)})
```

```
   <?> «statement»
 defin :: Parser [VarDef]
 defin = do {reserved «int»; il <- commaSep1 varDef; symbol «;»; return il}
 varDef :: Parser VarDef
 varDef = do {v  <- identifier; mi <- option Nothing
                 (do {i<- brackets dimension; return (Just i)}) ;
                 return (v,mi) }
program :: Parser Stmt
program = do {spaces; r <- stmt; eof; return r}
parseProgram :: String -> Program
parseProgram s = case parse program «» s  of
          {Left er -> error (show er); Right p -> p}
```

Контекстні умови

На жаль, повний опис деяких складних конструкцій процедурної мови контекстно-вільним синтаксисом (конкретним чи абстрактним) дати не можна. В цьому випадку використовують контекстно залежні правила, що описують необхідні додаткові умови.

Контекстні умови є предикатами, які накладають на об'єкти, визначені правилами абстрактного синтаксису, додаткові контекстно залежні обмеження. В подальшому при заданні семантики вважають, що використовують лише об'єкти, які задовольняють ці обмеження. Більшість контекстних умов пов'язана з правильним уживанням даних у програмі.

Для зберігання інформації про дані програми (Sctp) використовують «статичне» середовище (EnvSt)

```
data Sctp    = Ar | Sc deriving (Show, Eq)
type EnvSt =  [(String, Sctp)]
```

Використаному в контекстних умовах середовищу env відповідає список усіх, відомих у цьому місці програми, імен змінних. Ідентифікатор id — ім'я скалярної змінної, якщо в списку є пара (id, Sctp), або ім'я змінної-масиву, якщо в списку є пара (id, Ar). Контекстні умови задаємо для об'єктів: Program (iswfProgram), Stmt (iswfStmt) і Expr (iswfExpr). Функція iswfExpr, що перевіряє контекстні умови виразу Expr, повертає значення Maybe Sctp.

Якщо iswfExpr e env == Nothing, то контекстні умови для виразу e в статичному середовищі env не виконуються. Якщо iswfExpr e env == Just st, то умови виконуються і значення виразу або скалярне (st==Sc) або масив (st==Ar). Контекстні умови використовують допоміжний предикат: distinct is який перевіряє, що список is не має дублікатів.

```
distinct :: Eq a => [a] -> Bool
distinct []       = True
distinct (v:vs) = notElem v vs  &&  distinct vs
iswfProgram :: Program -> Bool
iswfProgram pr = iswfStmt pr []
iswfStmt :: Stmt -> EnvSt -> Bool
iswfStmt (Block vdl sl) env =
    (distinct (map fst vdl)) &&
    (let env1 = [(v,(maybe Sc (\_->Ar)) d) | (v,d) <- vdl] ++ env
     in all (flip iswfStmt env1) sl)
iswfStmt (Assign (v, Just ei) e) env =
    (lookup v env == Just Ar) &&
   (iswfExpr ei env == Just Sc) && (iswfExpr e env == Just Sc)
iswfStmt (Assign (v,Nothing) e) env =
    (lookup v env == Just Sc) && (iswfExpr e env == Just Sc)
iswfStmt (Read (v, Just ei)) env  =
     (lookup v env == Just Ar) && (iswfExpr ei env == Just Sc)
iswfStmt (Read (v, Nothing)) env  = (lookup v env == Just Sc)
iswfStmt (Write e) env                = (iswfExpr e env == Just Sc)
iswfStmt (If e s) env   =
      (iswfExpr e env == Just Sc) && iswfStmt s env
iswfStmt (While e s) env   =
     (iswfExpr e env == Just Sc) && iswfStmt s env
iswfExpr :: Expr -> EnvSt -> Maybe Sctp
iswfExpr (VarOp (v,Just e)) env  =
```

```
        if (lookup v env == Just Ar) && (iswfExpr e env == Just Sc)
                    then Just Sc else Nothing
iswfExpr (VarOp (v, Nothing)) env = lookup v env
iswfExpr (Const _) _                  = Just Sc
iswfExpr (BinOp _ e1 e2) env     =
   if (iswfExpr e1 env == Just Sc) && (iswfExpr e2 env == Just Sc)
                    then Just Sc else Nothing
```

Денотати

Завданням денотаційної семантики мови програмування є пов'язати з конструкціями мови певні математичні об'єкти (списки, таблиці, функції) —денотати, які задають «значення» (семантику) конструкцій. Семантика мови — набір семантичних функцій, які відображають синтаксичні конструкції мови, зокрема програму, у відповідні денотати.

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

Для опису семантики мови використовують стан — дане типу Work. Це кортеж із трьох елементів (inp, stg, out), що моделює три набори значень, із якими працює програма: список цілих [Integer] inp містить вхідні дані (файл введення); список значень [Maybe Integer] stg зберігає поточні значення всіх змінних програми (пам`ять); список цілих [Integer] out містить результуючі дані (файл виведення). Зауважимо, що значення скалярної змінної програми — це елемент списку stg, а значення масиву розмірності k — це k послідовно розташованих елементів 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 ніякого значення, то поточне значення відображають як Nothing.

Для роботи зі станом семантичні функції використовують функції, які управляють розміром другої компоненти робочого стану allocate, getBase і free; працюють зі значенням змінних getValue і updValue; працюють з 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) реалізує друга компонента 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, додає в список stg k елементів для розміщення значень локальних змінних блоку, а друга після закінчення виконання блоку вилучає їх. (Значення k визначається розмірностями локальних масивів і кількістю локальних скалярів. Наприклад для означення **int** i, c[10], m; маємо k=12.)

Функція getBase (inp,stg,out) повертає довжину списку stg. Якщо список stg, перед виконанням блоку, який вводить локальні змінні **int** i, c[10], m; має b (результат функції getBase) елементів stg = [a1, ..., ab], то після виконання функції allocate список stg буде мати (12+b) елементів stg = [vm, vc9, ..., vc1, vc0, vi, a1, ..., ab]. vci — позначає елемент списку stg, в якому буде зберігатися поточ-

не значення елементу c[i] локального масиву c в процесі виконання блоку. Коли необхідно при виконанні блоку отримати доступ до змінної vi з адресою 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 приводить до того, що в конкретних програмах одне ім'я може посилатися на різні значення (опис скалярної змінної у внутрішньому і у зовнішньому блоках). Крім того, одне ім'я масиву пов'язане з декількома значення (елементи з різними індексами).

Для пов'язування імен змінних з їхніми денотатами (номерами елементів списку stg, де зберігається поточне значення змінної) використовують середовище — значення типу Env.

```
type Env = [(String,(Maybe Int, Int))]
```

Якщо **int** a — скалярна змінна, з якою в env зв'язано елемент (a,(Nothing,3)), то stg[3] містить поточне значення a. Якщо **int** b[4] — масив, з яким в env зв'язано елемент (b,(Just 4,7)), то stg[7] містить поточне значення елемента b[0], stg[6] містить значення b[1], stg[5] — значення b[2], stg[4] — значення b[3] відповідно.

У семантичних функціях використовують такі допоміжні функції.

```
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»
alloc :: Int -> (Maybe Int) -> Int
alloc s mi = s+(maybe 1 id mi)
extEnv :: [VarDef] -> Int -> Env -> Env
extEnv vs b = \env ->
          let {(ns,mi) = unzip vs; mls  = zip mi (scanl alloc 0 mi);
                nenv1 = (zip ns [(m, (b+i+1)) | (m,i)<- mls]) ++ env}
           in nenv1
  getLocation :: String -> Env -> (Int,Int)
  getLocation s env = case fromJust $ lookup s env of
           {((Just n), k) -> (n,k);  (Nothing,  k) -> (1,k)}
  eLocation :: Var -> Env -> Work -> Int
  eLocation (s, Just ei) env = \w ->
           let {i  = fromIntegral (eExpr ei env w) ; (n,k) = getLocation s env}
           in if i>=0 && i<n then k+i else error "Index"
  eLocation (s, Nothing) env = \_w -> let (_,k) = getLocation s env in k
```

При обчисленні виразу використовують функцію applyBop op v1 v2, яка обчислює результат застосування бінарної операції op до цілих значень v1 і v2. У випадку обчислення операції Div або Mod, якщо другий операнд рівний нулю, то генерується помилка "DivOnZero" або "ModOnZero". Синтаксичний аналіз, обробляючи оголошення змінної, формує інформацію про її розмірність — об'єкт типу Maybe Int. Якщо змінна скаляр, то її розмірність — Nothing, а якщо масив із k елементів — то Just k. Функція alloc s mi за розміром s списку stg визначає його новий розмір після виділення пам'яті під значення змінної з розмірністю mi. (Друга компонента стану — список stg — реалізує динамічне управління пам'яттю: при вході в блок — на початку списку виділяється пам'ять, а при виході — звільнюється.) Функція extEnv vs ps b env за середовищем env, в якому потрібно виконати блок із локальними змінними vs і розміром b списку stg, який моделює пам'ять, будує нове розширене середовище.

Функція getLocation s env за іменем змінної s у середовищі env формує інформацію про частину списку stg, виділеного під зберігання значення змінної s, повертаючи пару (n,k): n — розмір участку (для скалярної змінної 1)і k — адреса його початку. Функція eLocation (s, mei) env за іменем змінної s і індексом mei вираховує адресу елемента пам'яті, де зберігається значення, в середовищі env. Якщо s — скалярна змінна, то mei == Nothing і адрес розміщення її значення getLocation s env. В іншому випадку mei == Just ei, за виразом ei вираховується значення індексу і. Якщо це значення задовольняє розмірність масиву, то вираховується адреса, в іншому випадку генерується помилка "Index".

Денотатами виразів Expr і операторів Stmt 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 extEnv vs b env викликається в функції iStm при формуванні денотату блоку (Block vs sts). Функція виділяє пам'ять під значення оголошених у блоці локальних змінних vs і за поточним розміром b списку stg формує денотати локальних змінних vs, розширюючи середовище env. Більшість семантичних функцій для доступу до денотатів змінних використовують середовище як додатковий параметр.

```
eExpr :: Expr -> Env -> Work -> Integer
eExpr (VarOp var) env      = \w -> getValue (eLocation var env w) w
eExpr (Const v) _              = \_ -> v
eExpr (BinOp op e1 e2) env = \w ->
                        applyBo op (eExpr e1 env w) (eExpr e2 env w)
extEnv :: [VarDef] -> Int -> Env -> Env
extEnv vs b = \env ->
        let {(ns,mi) = unzip vs; mls  = zip mi (scanl alloc 0 mi);
             nenv1 = (zip ns [(m, (b+i+1)) | (m,i)<- mls]) ++ env}
        in nenv1
iStmt :: Stmt -> Env -> Work -> Work
iStmt (Assign var e) env = \w ->
            updValue (eLocation var env w)(eExpr e env w) 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 (Block vs sts) env      = \w ->
             let {k  = foldl alloc 0 (map snd vs); w1 = allocate k w;
                  nenv = extEnv vs (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 = updValue (eLocation var env w) v  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

Використавши мову Haskell [3], наведено денотаційний опис імперативної мови, що містить цілі скалярні змінні і цілі одновимірні масиви. Опис охоплює синтаксис і денотаційну семантики мови.

Синтаксис містить: конкретний синтаксис, який описується синтаксичними правилами в розширеній БНФ; абстрактний синтаксис, що містить типи бінарної операції Op, змінної Var, виразу Expr, оголошення змінної VarDef, оператору Stmt і програми Program; контекстні умови, які накладають на об`єкти, визначені правилами абстрактного синтаксису, додаткові контекстно залежні обмеження.

Денотаційна семантика охоплює: основу денотатів мови — тип Work, що задає стан програми мови з масивами, та базові функції, які працюють із ним; денотати синтаксичних конструкцій мови (Expr, Stmt, Program) — функції зміни стану Work -> Integer, Work -> Work і [Integer] -> [Integer]; семантичні функції, які будують денотати синтаксичних конструкцій: eExpr, iStmt, iProgram.

Зауважимо, що всі функції базові, денотати і семантичні — чисті функції.

Використавши чисту функцію parseProgram, яка реалізує синтаксичний аналіз мови, будуємо інтерпретатор interpret (тобто реалізацію) мови. Якщо програма (рядок s) синтаксично правильна, то інтерпретатор, застосувавши до результату синтаксичного аналізу pr семантичну функцію iProgram, будує денотат програми, який потім застосовує 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).

Для зручності роботи будуємо основну функцію interpretFile sf ix, яка на початку вводить програму мови з файлу sf, а потім використовує чисту функцію interpret для інтерпретації програми.

```
    interpret :: String -> [Integer] -> [Integer]
interpret st ix = let {pr = parseProgram st; wf = iswfProgram pr}
                  in if wf then iProgram pr ix  else error «Contex»
interpretFile :: String -> [Integer] -> IO()
interpretFile sf ix = do {st <- readFile sf; print (interpret st ix)}
```

Найпростіший спосіб реалізації, враховуючи невеликий об'єм наведених Haskell-функцій, зібрати всі елементи в одному файлі DenotArFull.hs і скористатися інтерпретатором Haskell ghci.

Наведемо структуру файлу DenotArFull.hs.

```
{-# OPTIONS_GHC -Wall #-}
module DenotArFull where
import Text.ParserCombinators.Parsec
-- Абстракний синтаксис і типи,
-- що використовуються: Work, Sctp, EnvSt, Env
......
-- Синтаксичні аналізатори і Конкретний синтаксис БНФ в коментарях
.....
-- Контекстні умови
.......
-- Семантичні функції
.......
-- Інтерпретація
......
```

Далі наведено приклад роботи з програмою DenotArFull.hs, яка інтерпретує програму сортування (файл bubbleSort.txt), в командному рядку (програма cmd.exe). Програму cmd потрібно виконувати в каталозі, що містить файли DenotArFull.hs і bubbleSort.txt.

```
> ghci DenotArFull.hs
.....
ghci> interpretFile «bubbleSort.txt» [45,2,4,78,12,45,78,13,67,20]
[2,4,12,13,20,45,45,67,78,78]
```

Список літератури

1. Проценко В. С. Опис імперативної мови програмування у Haskell / В. С. Проценко // Наукові записки НаУКМА. Комп'ютерні науки. — 2021. — Т. 4. — С. 72–77.
2. Information technology. syntatic metalanguage. extended BNF. 1996.
3. Kurt W. Get programming with Haskell / W. Kurt. — Manning Publications, 2018.

References

Information technology. syntatic metalanguage. extended BNF (ISO/IEC 14977). (1996).

Kurt,W. (2018). *Get programming with Haskell.* Manning Publications.

Protsenko, V. S. (2021) Opys imperatyvnoi movy prohramuvannia u Haskell. *NaUKMA Research Papers. Computer Science, 4*, 72–77.

V. Protsenko

DENOTATIONAL SEMANTICS OF ONE-DIMENSIONAL ARRAYS

An imperative programming language is considered that has integer scalar data and one-dimensional arrays. Each variable a in this language is either an integer scalar variable a or an integer one-dimensional array a[k], where k>0 is a positive integer. Such an array has k elements, which are sequentially arranged and numbered as a[0], a[1], ..., a[k-1]. A program is a separate statement, usually a block with a

description of local integer variables (scalar or arrays) and a list of statements – the body of the block. All expressions in the statements calculate a scalar integer value. Work with an array is carried out only element by element. If a[k] is an entire one-dimensional array, then access to an individual element of the array is performed by an indexing operation - a record of the form a[e]. The value of the integer expression e – must be an integer from 0 to k-1.

In a program, one name can refer to different values (a variable description in the inner and outer blocks). In addition, one array name is associated with multiple values (elements with different indices). To associate variable names with their denotations (memory addresses where the current value of the variable is stored), an environment is used – a value of type Env.

type Env = [(String,(Maybe Int, Int))]

If Int a is a scalar variable with the denotation (Nothing,3) associated with it in the environment, then 3 is the memory address containing the current value of a. If Int b[4] is an array with which the denotation (Just 4,7) is associated, then 7 is the memory address containing the current value of element b[0], 6 is the address where the value b[1] is found, 5 is the address of b[2], and 4 is the address of b[3], respectively.

The functional language Haskell is used to describe this imperative language, which includes syntax and denotational semantics. All functions included in the description are pure. It is shown how to use these functions to build a pure function – a language interpreter.

Keywords: programming language, program, statement, expression, variable, one-dimensional array, syntax, denotation, semantic function, Haskell, interpreter.

Матеріал надійшов 25.06.2025