УДК 519.682.1
DOI: 10.18523/2617-3808.2021.4.72-77

Проценко В. С.

ОПИС ІМПЕРАТИВНОЇ МОВИ ПРОГРАМУВАННЯ У HASKELL

Розглянуто використання функціональної мови програмування Haskell як метамови. Наведено повну формальну специфікацію простої імперативної мови програмування з цілими даними, блочною структурою та операторами: присвоєння, введення, виведення, циклу і умовний. На основі специфікації побудовано інтерпретатор мови програмування.

Ключові слова: мова програмування, синтаксис, денотат, семантична функція, Haskell, синтаксичний аналіз, інтерпретатор.

Загальний метод опису мов програмування

При створенні мови програмування необхідно визначити її синтаксис і семантику. Основним завданням синтаксису є опис усіх конструкцій, що утворюють елементи мови. Для цього використовують конкретний синтаксис, який виділяє синтаксично правильні послідовності символів алфавіту мови. Найчастіше це скінченний набір правил, що породжують нескінченну множину всіх конструкцій мови, наприклад розширена нотація Бекуса – Наура (БНФ) [1].

Для опису семантики мови перевагу віддають абстрактному синтаксису, який у реальних мовах програмування коротший і більш наочний, ніж конкретний. При використанні мови Haskell [2] елементи абстрактного синтаксису – це просто типи даних, які задають структуру синтаксичних конструкцій. Зв'язок між об'єктами абстрактного синтаксису і конкретним синтаксисом програми у компіляторах розв'язує фаза синтаксичного аналізу.

Загальний метод опису мов програмування можна подати схемою:

КОНКРЕТНИЙ / АБСТРАКТНИЙ СИНТАКСИС – мова

ДЕНОТАТИ – припускаються

ЗНАЧЕННЯ (РЕАЛІЗАЦІЯ) – мова -> денотат.

Базою цього методу є денотаційна семантика, у якій семантику мова описують так. Спочатку фіксують денотати найпростіших синтаксичних об'єктів. Потім із кожною складеною синтаксичною конструкцією пов'язують семантичну функцію, яка за денотатами компонентів конструкції обчислює її 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.

Мова цілих виразів

Розглянемо застосування загального методу до опису мови цілих виразів. Ця мова містить вирази, що будуються з цілих чисел, знаків арифметичних операцій додавання +, віднімання -, множення *, цілочислового ділення /, визначення остачі % та круглих дужок. Її конкретний синтаксис у розширеній БНФ задається синтаксичними правилами, що містять конструкції: виразу *expr*, доданку *term*, множника *factor* і цілого числа *decimal*, і має вигляд:

expr = term, {*addOp*, *term*};
addOp = '+' | '_';
term = factor , {*mulOp*, *factor*};
mulOp = '*' | '/' | '%';
factor = decimal | '(' , *expr* , ')';
decimal = digit, {*digit*};
digit = '0'| '1'| '2'| '3'| '4'| '5'| '6'| '7'| '8'| '9';

У синтаксичних правилах уже враховано пріоритети операцій +, - і *, /, % та їхню ліву асоціативність.

Абстрактний синтаксис визначається типами: операції *Op* та вирази *Expr*.

***data** Op = Plus | Minus | Times | Div | Mod **deriving** (Show, Eq)*

***data** Expr = BinOp Op Expr Expr | Const Integer **deriving** (Show, Eq)*

Областю денотатів є тип *Integer*. Значення задається функцією *eExpr*, яка використовує допоміжну функцію *applyBo*. Їхні типи й означення мають вид:

eExpr :: Expr -> Integer
eExpr (Const v) = v
eExpr (BinOp bo e1 e2) = applyBo bo (eExpr e1) (eExpr e2)

```
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»
```

Наприклад, абстрактним синтаксисом цілого виразу 5 + 4 * 3 є вираз типу *Expr*

```
ex1 :: Expr
ex1 = BinOp Plus (Const 5) (BinOp Times
(Const 4) (Const 3))
```

Його денотат обчислюється функцією *eExpr*:

```
eExpr ex1  =  17
```

У Haskell можна ефективно реалізувати синтаксичний аналіз виразу, використавши, наприклад, бібліотеку синтаксичних аналізаторів parsec. За посиланням https://github.com/ProtsenkoVS/lngExpr.git можна завантажити простий проєкт lngExpr у Haskell. Проєкт описує мову виразів, зокрема синтаксичний аналізатор *parseExpr st*, який за рядком *st*, що містить цілий вираз, будує його абстрактний синтаксис – об'єкт типу Expr.

Просте з'єднання двох чистих функцій *parseExpr* і *eExpr* будує інтерпретатор мови цілих виразів *interpret*

```
interpret :: String -> String
interpret st = let e = parseExpr st
               in show $ eExpr e
```

Мова зміни стану

Розглянемо імперативну мову програмування LS, що має цілі скалярні дані. Програма в цій мові будується із локальних цілих змінних та операторів, що обробляють дані (змінні й константи). Застосовують прості і складні (структурні) оператори: присвоєння *v := ex*, введення ***read*** *v*, виведення ***write*** *ex*, умовний ***if*** *ex* ***then*** *st*, циклу ***while*** *ex* ***do*** *st*, блок {***int*** *v1,..,vn; st1; ...;stk*}, де *v*, *v*1, …, *vn* – змінні, *ex* – вираз, *st*, *st*1, …, *stk* – оператори. У виразах використовують константи, змінні й операції +, -, *, /, %.

Далі наведено приклад простої програми мовою LS:

```
{ int b, e, out;
  read b; read e; out:= 0;
  if (b)   if (e)
        {int i; i:=0; out := 1;
        while (e-i) {out := out*b; i := i+1} };
  write out
}
```

Програма вводить два цілі значення *b* і *e*. Якщо вони додатні, то в змінній *out* обчислюється значення *b* у степені *e*, в інших випадках її значення є 0. Значення *out* в кінці виводиться.

Синтаксис

Конкретний синтаксис конструкцій мови – програми *program*, оператора *stmt*, означення змінних *defin*, виразу *expr*, доданка *term*, множника *factor*, операцій додавання *addOp*, операцій множення *mulOp*, змінної *identifier* і цілого *decimal* – описують такими синтаксичними правилами:

```
program = stmt ;
stmt = 'while', '(', expr, ')', stmt;
             | 'if', '(', expr, ')', stmt;
             | 'read', identifier;
             | 'write', expr;
             | identifier, ':=', expr;
             | '{', [defin], stmt, {';', stmt}, '}';
defin = 'int' identifier, { ',', identifier}, ';' ;
expr = term, {addOp , term} ;
term = factor, {mulOp , factor} ;
factor = decimal | '(', expr , ')'| identifier;
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';
```

Абстрактний синтаксис визначається типами: програми *Program*, оператора *Stmt*, операції *Op* та виразу *Expr*.

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

Денотати

Для опису семантики мови LS – денотатів виразів і операторів – використовують робочий стан *Work* – кортеж із трьох компонент *(inp, mem, out)*: *inp* – список цілих *[Integer]* задає вхідні дані (файл введення), *mem* – список пар *[(String, Integer)]*, який зв'язує з ідентифікатором змінної

(*String*) її поточне значення (*Integer*), *out* – список цілих *[Integer]* задає результуючі дані (файл виведення).

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

Із робочим станом *Work* працюють такі функції:

```
getValue:: String -> Work -> Integer
getValue s = \(_,mem,_) -> case lookup s mem of
                Just v -> v
                Nothing -> error «getValue»
```

getValue s (inp, mem, out) – вибирає зі списку пар *mem* поточне значення змінної *s*, якщо відповідної пари немає, то генерується помилка *"getValue"*

```
updValue :: String -> Integer -> Work -> Work
updValue s v = \(inp,mem,out) -> (inp, update mem s v, out)
where update :: [(String, Integer)] -> String -> Integer -> [(String, Integer)]
      update [] _ _ = error «updValue»
      update ((x,_):sx) s1 v1 | x==s1 = (x,v1): sx
      update (x:sx) s1 v1 = x:(update sx s1 v1)
```

updValue s v (inp, mem, out) – змінює робочий стан, встановлюючи у списку пар *mem* – поточне значення змінної *s* рівним *v*. Якщо відповідної пари немає, то генерується помилка *"updValue"*.

```
writeValue :: Integer -> Work -> Work
writeValue v = \(inp, mem, out) -> (inp, mem, out ++ [v])
```

writeValue v (inp, mem, out) – змінює робочий стан, додаючи в кінець результуючого списку *out* ціле значення *v*.

```
extMemory :: [String] -> Work -> Work
extMemory vs = \(inp,mem,out) -> (inp,[(v,0) | v <- vs] ++ mem,out)
```

extMemory vs (inp, mem, out) – змінює робочий стан, додаючи на початок списку пар *mem* всі цілі змінні зі списку *vs*. З кожною змінною зв'язується початкове значення 0.

```
dropMemory :: Int -> Work -> Work
dropMemory t = \(inp,mem,out) -> (inp, drop t mem,out)
```

dropMemory t (inp, mem, out) – змінює робочий стан, вилучаючи зі списку пар *mem t* перших пар. Функції *extMemory* і *dropMemory* працюють зі списком *mem* як зі стеком: перша – на початку виконання блоку додає змінні в *mem*, а друга – по закінченню блоку вилучає їх.

```
readInput :: Work -> Integer
readInput = \(inp,_,_) -> if null inp then error «readInput» else head inp
```

readInput (inp, mem, out) – повертає перше значення з списку *inp*; якщо список порожній, то генерується помилка *"readInput"*

```
dropInput :: Work -> Work
dropInput = \(inp,mem,out) -> (tail inp,mem,out)
```

dropInput (inp, mem, out) – змінює робочий стан, вилучаючи перший елемент списку *inp.* Її використовують, як правило, після функції *readInput*.

Семантичні функції

Денотатами виразів та операторів мови LS є, відповідно, функції зміни робочого стану типу: *Work -> Integer* і *Work -> Work*, денотатом програми – функція типу: *[Integer] -> [Integer]*.

```
eExpr :: Expr -> Work -> Integer
eExpr (Var s) = \w -> getValue s w
eExpr (Const v) = \_ -> v
eExpr (BinOp op e1 e2) = \w -> applyBo op (eExpr e1 w) (eExpr e2 w)
iStmt :: Stmt -> Work -> Work
iStmt (Assign var e) = \w -> updValue var (eExpr e w) w
iStmt (If e s) = \w -> if eExpr e w > 0 then iStmt s w else w
iStmt wh@(While e s) = \w -> if eExpr e w > 0 then iStmt wh (iStmt s w) else w
iStmt (Block vs sts) = \w -> let w1 = extMemory vs w
                                 w2 = foldl (flip iStmt) w1 sts
                             in dropMemory (length vs) w2
iStmt (Read var) = \w -> let v = readInput w
                         in  updValue var v (dropInput w)
iStmt (Write e) = \w -> writeValue (eExpr e w) w
iProgram :: Program -> [Integer] -> [Integer]
iProgram prog ix = let w = (ix, [],[])
                       (_,_,ox) = iStmt prog w
                   in ox
```

Семантичні функції *eExpr*, *iStmt* і *iProgram* обчислюють відповідно денотати конструкцій *Expr*, *Stmt* і *Program*.

На початку виконання програми перша компонента робочого стану *Work* містить список вхідних даних, а друга і третя компонента – порожні списки. По закінченню виконання програми значення третьої компоненти – її результат.

Синтаксичний аналіз

Haskell має декілька бібліотек, що будують синтаксичні аналізатори, одна з них – бібліотека parsec. Бібліотека parsec надає можливість ефективно реалізувати синтаксичний аналіз мови LS.

```
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
```

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

Лексична основа в більшості мов програмування схожа. Тому для створення лексичних аналізаторів спочатку описують лексичну основу мови – запис типу *LanguageDef*, який містить інформацію про коментарі, ідентифікатори, зарезервовані слова тощо. Часто для цього використовують функцію *emptyDef*, яка розширює загальну мінімальну лексичну основу необхідними елементами.

lexDef – описує лексичну основу, яка не допускає ніяких коментарів, задає списки зарезервованих слів і операторів. Ідентифікатори починаються з букви (*identStart = letter*) і містять лише букви і цифри (*identLetter = alphaNum*).

```
lexer :: P.TokenParser ()
lexer =  P.makeTokenParser lexDef
```

Функція *P.makeTokenParser*, використовуючи лексичну основу *lexDef*, будує запис типу *P.TokenParser*, який містить 29 лексичних аналізаторів, що їх можна використовувати при побудові інших аналізаторів. Кожний із лексичних аналізаторів вибирається відповідним селектором. Наприклад: *P.decimal lexer* – аналізатор, що розпізнає додатне ціле число в десятковій системі і повертає його значення; *P.reserved lexer "while"* – аналізатор, що розпізнає зарезервоване слово ***while***, перевіряючи, що слово не є початком правильного ідентифікатора; *P.identifier lexer* – аналізатор, який розпізнає правильний ідентифікатор, але не допускає зарезервовані слова.

Синтаксичний аналіз виразів реалізують такі функції:

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

factorL – аналізатор, що розпізнає простий вираз-множник *factor*, який може бути: виразом у дужках, числом або змінною (ідентифікатор).

```
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)
```

table – таблиця синтаксичних аналізаторів усіх бінарних операцій, що використовуються у виразах мови LS. Кожний елемент таблиці – це список аналізаторів операції одного пріоритету. З кожним аналізатором вказується асоціативність операції (есі лівоасоціативні *E.AssocLeft*). Елементи таблиці впорядковані згідно зі зменшенням пріоритету.

```
exprL :: Parser Expr
exprL = E.buildExpressionParser table factorL
```

Кожний вираз у мові будується із простих виразів-множників та бінарних операцій згідно з їхніми пріоритетами та асоціативністю. Функцію *buildExpressionParser* будує синтаксичний аналізатор виразів для простих виразів, що розпізнаються аналізатором *factorL*, з операціями, які розпізнають аналізатори таблиці *table*.

Наступні функції будують аналізатори, що розпізнають оператори і програму:

```
definL :: Parser [String]
definL = P.reserved lexer «int» *>
         P.commaSep1 lexer (P.identifier lexer)
<* P.symbol lexer «;»
```

definL – аналізатор, котрий розпізнає об'яву змінних на початку блоку (конструкція виду ***int*** v1,..,vn;).

```
stmtL :: Parser Stmt
stmtL =
    (P.reserved lexer «while» >> While <$> P.parens lexer exprL <*> stmtL)
    <|> (P.reserved lexer «if» >> If <$> P.parens lexer exprL <*> stmtL)
    <|> (P.reserved lexer «read» >> Read <$> P.identifier lexer)
    <|> (P.reserved lexer «write»>> Write <$> exprL)
    <|> (P.identifier lexer >>=
        (\var -> P.symbol lexer «:=» >> Assign var <$> exprL))
    <|> (P.braces lexer (Block <$> (option [] definL)
                <*> P.semiSep1 lexer stmtL))
```

stmtL – аналізатор, що розпізнає один із шести операторів мови: циклу ***while*** ex ***do*** st, умовний ***if*** ex ***then*** st, введення ***read*** v, виведення ***write*** ex, присвоєння v := ex або блок {***int*** v1,..,vn; st1; …;stk}, формуючи як результат – одне з можливих значень типу Stmt. При цьому використовується декілька раз комбінатор вибору (<|>).

```
programL :: Parser Program
programL = P.whiteSpace lexer *> stmtL <* eof
```

Аналізатор *programL* виконує синтаксичний аналіз програми, враховуючи, що програма може

мати проміжки до і після оператора, який її складає, і нічого більше.

```
parseLSL :: String -> Program
parseLSL s = case parse programL «» s of
               Left _ -> error «Syntax»
              Right stmt  -> stmt
```

parseLSL s – реалізує синтаксичний аналіз програми мови LS у рядку *s*, повертаючи *stmt* – абстрактний синтаксис програми, заданої в рядку, або генерує повідомлення про синтаксичну помилку *"Syntax"*.

Інтерпретатор

Опис імперативної мови програмування LS, що використовує мову Haskell, містить:

- конкретний синтаксис, який описують 12 синтаксичних правил у розширеній БНФ;
- абстрактний синтаксис – типи *Op*, *Expr*, *Stmt* і *Program*;
- денотати мови, основу яких складають тип *Work*, що описує робочий стан програми мови LS, і базові функції, що працюють із ним (*getValue, updValue, writeValue, extMemory, dropMemory, readValue, dropValue*). Денотати мови – це функції типу*: Work -> Integer* для *Expr*, *Work -> Work* для *Stmt* і *[Integer] -> [Integer]* для *Program*;
- семантичні функції, що будують денотати: *eExpr*, *iStmt* і *iProgram*.

Зауважимо, що всі функції – базові, що працюють із робочим станом *Work*, денотати і семантичні – є чистими.

Використавши чисту функцію parseLSL, що реалізує синтаксичний аналіз мови програмування LS, можна побудувати інтерпретатор (тобто реалізацію) мови LS:

```
interpret :: String -> [Integer] -> [Integer]
interpret st ix = let pr = parseLSL st
                      den =  iProgram pr
                  in den ix
```

Синтаксичний аналізатор *parseLSL* виконує синтаксичний аналіз програми в рядку *st*, будуючи її абстрактний синтаксис *pr*. Семантична функція *iProgram* будує за програмою *pr* її денотат *den*. Функція *den* застосовується до вхідних даних *ix*, інтерпретуючи (виконуючи) початкову програму *st*.

Інтерпретатор *interpret* – чиста функція, але якщо на кроці синтаксичного аналізу або застосування денотату виникає помилка, то обчислення переривається, генеруючи відповідне повідомлення.

За посиланням https://github.com/ProtsenkoVS/lngState.git можна завантажити проєкт lngState, який містить повний опис імперативної мови програмування SPL у Haskell, зокрема різні варіанти синтаксичного аналізу і семантичних функцій. Проєкт забезпечує роботу з інтерпретатором у режимі REPL, введення програми з файлу, оброблення переривань і зручний вибір різних варіантів синтаксичного аналізу і семантичних функцій.

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

DESCRIPTION OF THE IMPERATIVE PROGRAMMING LANGUAGE IN HASKELL

When creating a programming language, it is necessary to determine its syntax and semantics. The main task of syntax is to describe all constructions that are elements of the language. For this purpose, a specific syntax highlights syntactically correct sequences of characters of the language alphabet. Most often it is a finite set of rules that generate an infinite set of all construction languages, such as the extended Backus-Naur (BNF) form.

To describe the semantics of the language, the preference is given to the abstract syntax, which in real programming languages is shorter and more obvious than specific. The relationship between abstract syntax objects and the syntax of the program in compilers solves the parsing phase.

Denotational semantics is used to describe semantics. Initially, it records the denotations of the simplest syntactic objects. Then, with each compound syntactic construction, a semantic function is associated, which by denotations of components of a design calculates its value – denotation. Since the program is a specific syntactic construction, its denotation is possible to determine using the appropriate semantic function. Note that the program itself is not executed when calculating its denotation.

The denotative description of a programming language includes the abstract syntax of its constructions, denotations – the meanings of constructions and semantic functions that reflect elements of abstract syntax (language constructions) in their denotations (meanings).

The use of the functional programming language Haskell as a metalanguage is considered. The Haskell type system is a good tool for constructing abstract syntax. The various possibilities for describing pure functions, which are often the denotations of programming language constructs, are the basis for the effective use of Haskell to describe denotational semantics.

The paper provides a formal specification of a simple imperative programming language with integer data, block structure, and the traditional set of operators: assignment, input, output, loop and conditional.

The ability of Haskell to effectively implement parsing, which solves the problem of linking a particular syntax with the abstract, allows to expand the formal specification of the language to its implementation: a pure function — the interpreter.

The work contains all the functions and data types that make up the interpreter of a simple imperative programming language.

Keywords: programming language, syntax, denotation, semantic function, Haskell, parsing, interpreter.

Матеріал надійшов 08.06.2021