Семантичний аналіз

Юрій Стативка

Жовтень, 2024 р.

Зміст

1 Приклад статично типізованої мови

В прикладі синтаксичного аналізатора, див. [1], ми розглянули власне перевірку синтаксису для динамічно типізованої мови. Перехід до семантичного аналізу зручно розглядати для мови, типізованої статично.

1.1 Граматика

Нехай мова визначається граматикою з секцією оголошень з визначеними типами `int` і `float`

```
Program       = program
                    DeclSection
                begin  StatementList  end

DeclSection   = var DeclarList

DeclarList    = {Ident '::' Type ';'}

Type          = int | float

StatementList = Statement  { Statement }

Statement = Assign
          | IfStatement
          | Read
          | Write

Assign = Ident ':=' Expression

Expression = Term {( '+' | '-' ) Term}

Term = Factor {( '*' | '/' ) Factor}

Factor = Id
       | Const
       | '(' Expression ')'

IfStatement = if BoolExpr then Statement else Statement endif


BoolExpr = Expression ('='|'<='|'>='|'<'|'>'|'<>') Expression
         | true
         | false

Read = read '(' Ident ')'

Write = write '(' Ident ')'
```

Правила для нетерміналів, оброблені лексичним аналізатором, несуттєві для подальшого розгляду і наведені тут тільки для повноти граматики.

```
Ident = Letter {Letter | Digit }

Const = Float | Int

Float = Digit {Digit} '.' {Digit}
```

```
Int = Digit {Digit}

Letter = a | b | c | d | e | f | g | h | i | j | k | l | m | n | o
         | p | q | r | s | t | u | v | w | x | y | z

Digit  = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

WhiteSpace = (Blank|Tab) {Blank|Tab}

Blank  = ' '

Tab    = '\t'

Newline = '\n'
```

Зауважимо, що для кожної змінної передбачена окрема декларація, а кожна `if` - альтернатива містить тільки один оператор `(statement)`. Списки введення і виведення – тільки по одній змінній.

1.2 Семантика мови

Вважатимемо, що специфікація мови передбачає наступну семантику[1]:

1. Тип кожної змінної має бути визначений у розділі оголошень.
2. Повторне оголошеної змінної викликає помилку на етапі (у фазі) компіляції.
3. Використання неоголошеної змінної (`l-value` чи `r-value`), викликає помилку на етапі компіляції.
4. Змінна набуває значення в інструкції присвоювання `Assign` або в інструкції введення `Read`.
5. Використання змінної (`r-value`), що не набула значення, викликає помилку.
6. Всі оператори у цій мові – бінарні. Типи лівого і правого операндів мають бути однаковими (`int або float` і для арифметичних операторів, і для операторів відношення).
7. Оператор `'/'`, застосований до операндів типу `int` означає цілочисельне ділення.
8. Тип результату застосування арифметичних операторів збігається з типом операндів.
9. Результат застосування операторів відношення має значення типу `bool`.

[1]Про семантику мов програмування див. також [2].

10. В операторі присвоювання `Assign = Ident ':=' Expression` тип змінної з ідентифікатором `Ident` повинен збігатись з типом `Expression`.

11. Ділення на нуль викликає помилку.

Тоді коректна програма представленою мовою може виглядати, наприклад, так.

```
program
var
  a :: int;
  b :: float;
  x :: float;
begin
  read(a)
  x:= 12.4 + 2.0
  if  a + 1 < 3
    then b := 12.34 + ((1.0+x)*(x/2.1+3.5))
    else b := x
  endif
  write(b)
end
```

1.3 Синтаксичний аналізатор

Синтаксичний аналізатор цієї мови, порівняно з прикладом із [1], очевидно зводиться до додавання виклику `parseDeclSection()` у функції `parseProgram()` та визначення як самої функції `parseDeclSection()`, так і функцій, які вона викликає/активує.

```
def parseProgram():
    try:
        # перевірити наявність ключового слова 'program'
        parseToken('program','keyword','')
        # перевірити коректність розділу декларацій
        parseDeclSection()
        # перевірити наявність ключового слова 'program'
        parseToken('begin','keyword','')
        # перевірити синтаксичну коректність списку інструкцій StatementList
        parseStatementList()
        # перевірити наявність ключового слова 'end'
        parseToken('end','keyword','')
        # повідомити про синтаксичну коректність програми
        print('Parser: Синтаксичний аналіз завершився успішно')
        return True
    except SystemExit as e:
        # Повідомити про факт виявлення помилки
        print('Parser: Аварійне завершення з кодом {0}'.format(e))
```

2 Семантичний аналіз: обмеження

Семантичний аналіз виконуватимемо в процесі синтаксичного аналізу. Це означає, що розроблений парсер треба доповнити кодом для аналізу семантики.

У списку семантичних обмежень мови, див. розд. 1.2:

- пункти 1 і 2 здійснюються простою перевіркою під час розбору секції декларацій. Передбачається, що таблиця ідентифікаторів/змінних при синтаксичному розборі будується заново;
- пункт 3 передбачає перевірку наявності ідентифіктора у таблиці ідентифікаторів при розборі секції операторів (функція `parseStatementList()`);
- пункт 4 дає можливість фіксувати факт ініціювання змінної (набуття значення);
- пункт 5 передбачає необхідність перевірки факту ініціювання змінної при її використанні;
- пункти 6 -9 декларують або передбачають перевірку типів у виразах;
- пункт 10 передбачає необхідність перевірки типів при використанні оператора присвоювання ':=' ;
- пункт 11 передбачає перевірку того, що правий операнд оператора '/' не має конкретного значення (не дорівнює нулю).

2.1 Побудова таблиці ідентифікаторів/змінних і контроль унікальності оголошення

Приймемо, що таблиця змінних `tableOfVar` – це словник `id:(indx,type,value)`, порожній перед початком парсингу. Ключ словника – ідентифікатор. При оголошенні змінної до таблиці додається запис з декларованим типом і значенням, встановленим в `undefined`.

У відповідності до правил граматики

```
DeclSection   = var DeclarList
DeclarList    = {Ident '::' Type ';'}
Type          = int | float
```

розбір може бути реалізований, наприклад, так.

```
def parseDeclSection():
  parseToken('var','keyword','')
  parseDeclarList()
```

```
1 def parseDeclarList():
2   global numRow
3   numLine, lex, tok = getSymb()
4   while (lex, tok) not in [('begin','keyword')]:
```

```
5       # якщо це ідентифікатор
6       if tok == 'ident':
7         numRow += 1
8         parseToken('::','type_var','\t')
9         numLineT, lexT, tokT = getSymb()
10        numRow += 1
11        if lexT in ('int','float'):
12          procTableOfVar(numLine,lex,lexType,'undefined')
13          parseToken(';',';','\t')
14        else: failParse('невідомий тип',(numLineT, lexT, tokT))
15      else: failParse('очікувався ідентифікатор',(numLine, lex, tok))
16      numLine, lex, tok = getSymb()
```

Власне, перевірка того, що змінна не була оголошена раніше, здійснюється у рядку 12 при обробці таблиці змінних, хоча, звісно, таку перевірку можна було виконати уже у рядку 7.

```
def procTableOfVar(numLine,lexeme,type,value):
  indx=tableOfVar.get(lexeme)
  if indx is None:
    indx=len(tableOfVar)+1
    tableOfVar[lexeme]=(indx,type,value)
  else: failParse('повторне оголошення змінної', \
                        (numLine, lexeme,type,value))
```

2.2 Використання неоголошеної змінної

Перевіряти перед використанням змінної, чи була вона оголошена, можна в процесі контролю типів виразів та інструкцій. Наприклад, довідатись про тип змінної, або про те, що вона не була оголошена, можна за допомогою такої функції.

```
def getTypeVar(id):
  try:
    return tableOfVar[id][1]
  except KeyError:
    return 'undeclared_variable'
```

Якщо змінна з вказаним ідентифікатором є у таблиці змінних – повертається її задекларований тип. Якщо ні – повертається тип `undeclared_variable`.

2.3 Фіксація та перевірка факту ініціації змінної

При синтаксичному розборі, змінній з ідентифікатором `Ident` у функціях, що відповідають правилам граматики

```
Assign = Ident ':=' Expression
Read = read '(' Ident ')'
```

в таблиці змінних треба встановити значення `assigned` замість `undefined`.

Тоді перевірку ініціації оголошеної змінної можна реалізувати, наприклад так.

```
def isInitVar(id):
  try:
    if tableOfVar[id][2] == 'assigned':
      return True
    else:
      return False
  except KeyError:
    return 'undeclared_variable'
```

3 Семантичний аналіз: перевірка типів

З розгляду правил граматики, наприклад

```
Assign = Ident ':=' Expression
```

очевидно, що потрібен механізм визначення типів не тільки констант та змінних, але і виразів. У випадку вбудованого у парсер семантичного аналізатора зручно вважати, що функції типу `parseName()`, як-от `parseExpression()` повертає тип розібраного виразу. Якщо ж перевірка типів при виконанні `parseName()` завершується неуспіхом, то функція повертає тип `type_error`.

Однак функції, на кшталт `parseAssign()` чи `parseIfStatement()`, відповідальні за розбір та перевірку типів у інструкціях. Виконання інструкції, як відомо, не повертає жодного значення, яке би мало певний тип. Тому приймається, що такі функції повертають тип `void`, якщо перевірка типів при їх виконанні закінчилась успіхом, якщо неуспіхом – `type_error`.

3.1 Перевірка типів виразів

3.1.1 Змінні та константи

Дізнатись тип змінної можна як показано у розд. 2.2, а константи – за її токеном у таблиці констант, побудованої лексичним аналізатором, напр. так.

```
def getTypeConst(literal):
  # tableOfConst - словник {literal:(indx,type)}
  return tableOfConst[literal][1]
```

3.1.2 Вирази

Оскільки вираз – це послідовність операндів і операторів, то його тип визначається типом результату застосування операторів до своїх операндів.

Декларована у пунктах 6 -9 семантика, див. розд. 1.2, може контролюватись певною функцією, як-от `getTypeOp(lType,op,rType)`, де `lType ,op i rType` – лівий операнд, бінарний оператор та правий операнд відповідно.

```
def getTypeOp(lType,op,rType):
  # типи збігаються?
  typesAreSame = lType == rType
  # типи арифметичні?
  typesArithm  = lType in ('int','float') and \
                 rType in ('int','float')
  if   typesAreSame and \
       typesArithm  and \
       op in '+-*/'      : typeRes = lType
  elif typesAreSame and \
       typesArithm  and \
       op in ('<','<=','>','>=','=','<>'):
                           typeRes = 'bool'
  else:                    typeRes = 'type_error'
  return typeRes
```

Тоді, з огляду на правило граматики

```
Expression = Term {( '+' | '-' ) Term}
Term = Factor {( '*' | '/' ) Factor}
Factor = Id
       | Const
       | '(' Expression ')'
```

тип довільного арифметичного виразу може бути обчислений функцією `parseExpression()`, наприклад, так.

```
1  def parseExpression():
2    global numRow
3    lType = parseTerm()
4    F = True
5    # продовжувати розбирати Доданки (Term)
6    # розділені лексемами '+' або '-'
7    while F:
8      numLine, lex, tok = getSymb()
9      if tok in ('add_op'):
10       numRow += 1
11       rType = parseTerm()
12       resType = getTypeOp(lType,lex,rType)
13       if resType != 'type_error':
14        # продовжити обробку
15        ltype = resType
16       else:
17        tpl = (numLine,lType,lex,rType) #для повідомлення про помилку
18        failSem(resType,tpl)
19     else:
20         F = False
21   return resType
```

В рядку 3 обчислюється тип виразу – лівого аргументу, розібраного функцією `parseTerm()` зі збереженням результату у змінній `lType`. Якщо наступний символ – адитивний оператор `+` або `-` (рядки 8 -9), то з'ясувати тип правого операнда, рядок 11, та знайти тип результату
`resType = getTypeOp(lType,lex,rType)` (рядок 12).
Якщо `resType != 'type_error'` – продовжити обробку, яка у цьому прикладі, рядок 15, зводиться до того, що обчислений вираз вважатиметься тепер лівим операндом наступного адитивного оператора, якщо такий зустрінеться далі. Якщо ж `resType == 'type_error'`, то активується функція обробки помилок `failSem(resType,tpl)`, рядок 18. Функція `parseExpression()` завжди повертає тип `resType`, обчислений у рядку 12.

Подібним же чином мають бути організовані й інші функції, що викликаються із `parseExpression()` та функції для розбору виразів іншого типу, як-от `parseBoolExpr()` для логічних виразів у цій мові.

3.2 Перевірка типів інструкцій

Як уже зазначалось, функції синтаксичного аналізатора для розбору інструкцій, такі, наприклад, як `parseAssign()` чи `parseIfStatement()`, повертають тип `void`, якщо перевірка типів у відповідній конструкції програми закінчилась успіхом, або `type_error`, якщо неуспіхом.

3.2.1 Інструкція присвоювання

Правило перевірки типів, див. п. 10 у розд. 1.2, для інструкції присвоювання `Assign = Ident ':=' Expression` можна представити формально як

```
if type(Ident) == type(Expression)
  then resType = 'void'
  else resType = 'type_error'
```

Тут `type(X)` означає функцію визначення типу аргументу `X`.

У нашому випадку таке правило може бути зреалізоване мовою `python`, наприклад, так.

```
1  def parseAssign():
2    global numRow
3    numLine, lex, tok = getSymb()
4    numRow += 1
5    lType = getTypeVar(lex)
6    if lType == 'undeclared_variable':
7      failSem(lType,(numLine, lex))
8    parseToken(':=','assign_op','\t')
9    rType = parseExpression()
10   resType = getTypeOp(lType,':=',rType)
11   if resType == 'type_error':
12     failSem(resType,(numLine, lex,))
13   return resType
```

Рядки 6 -7 запобігають використанню неоголошеної змінної. Функцію `getTypeOp(lType,op,rType)`, визначену раніше для бінарних арифметичних операторів та операторів порівняння слід тепер доповнити альтернативою

```
elif typesAreSame and op in (':='):
    typeRes = 'void'
```

3.2.2 Інструкції введення/виведення

На етапі семантичного аналізу перевірка типів для інструкції введення `Read = read '(' Ident ')'` не може бути виконана без генерування додаткового коду, оскільки невідомо, який літерал буде введений користувачем. Однак це завдання може бути виконане, наприклад, на етапі генерування коду.

Можна вважати, як у нашому прикладі, де немає явно вказаних семантичних обмежень для інструкції виведення `Write = write '(' Ident ')'`, що `parseWrite()` повертає 'void', якщо `Ident` – оголошена та ініційована змінна. Якщо ні – повернути `type_error`. У випадку, якщо `Ident` не ініційована явно – можна, наприклад, видалити таку інструкцію на етапі оптимізації. З відповідним попередженням для користувача компілятора, звичайно.

3.2.3 Інші інструкції

Правило обчислення типу для інструкції циклу з передумовою `WhileStmnt = while BoolExpr do Stmnt` можна формально представити у формі

```
if type(BoolExpr) == 'bool'
  then resType = type(Stmnt)
  else resType = 'type_error'
```

Правило обчислення типу послідовності/списку інструкцій `Stmnt1; Stmnt2` – у формі

```
if type(Stmnt1) == 'void'  and type(Stmnt2) == 'void'
  then resType = 'void'
  else resType = 'type_error'
```

Сенс таких правил полягає, врешті, у тому, що перевірка типів має закінчитись успіхом при компіляції кожної інструкції.

Ґрунтовно про типи в мовах програмування та перевірку типів див. [3] та [4]

Література

[1] Матеріали до виконання роботи " Синтаксичний аналіз методом рекурсивного спуску".

[2] Стативка Ю.І. Формальні мови: Основні концепти і представлення [Текст]: навч. посіб. / Ю. І. Стативка. – Київ: КПІ ім. Ігоря Сікорського, 2023. – 87 с.

[3] Alfred Aho, Monica Lam, Ravi Sethi, Jeffrey Ullman Compilers: Principles, Techniques, and Tools. Addison Wesley; 2nd edition, 2006. – 1040 p.

[4] Benjamin C. Pierce: Types and Programming Languages. - The MIT Press, 2002. - 648 p.