**Синтаксичний аналiз методом рекурсивного спуску**

Варіант № 12

Арифметика: цілі та дійсні числа, основні чотири арифметичні операції (додавання, віднімання, ділення та множення), піднесення до степеня(правоасоціативна операція), дужки

Особливості: унарний мінус

Інструкція повторення: for <ід>=<вираз> by <вираз> while<відношення> do <оператор>

Інструкція розгалуження: if <відношення> {<список операторів>}

Повна граматика мови KK

Program = start ProgName ‘;’ StatementList

ProgName = Ident

*Алфавіт*

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’

SpecSsign = ’.’ | ’,’ | ’:’ | ’;’ | ’(’ | ’)’| ’=’ | ’+’ | ’-’ | ’\*’ | ’/’ | ‘^’ | ’<’ | ’>’| WhiteSpace | EndOfLine

WhiteSpace = ’ ’ | ’\t’

EndOfLine = ’\n’ | ’\r’ | ’\r\n’ | ’\n\r’

*Символи*

SpecSymbols = ArithOp | RelOp | BracketsOp | AssignOp | Punct

ArithOp = AddOp | MultOp | NeltOp

AddOp = ’-’

MultOp = ’\*’ | ’/’

NeltOp = ‘^’

RelOp = ’==’ | ’<= ’ | ’<’ | ’>’ | ’>=’ | ’<>’ | ‘!=’

BracketsOp = ’(’ | ’)’

AssignOp = ’=’

Punct = ’.’ | ’,’ | ’:’ | ’;’

*Індифікатор*

Ident = Letter {Letter | Digit }

*Конастанти*

Const = IntNumb | RealNumb | BoolConst

IntNumb = [Sign] UnsignedInt

RealNumb = [Sign] UnsignedReal

Sign = ’-’

UnsignedInt = Digit {Digit}

UnsignedReal = ’.’ UnsignedInt| UnsignedInt ’.’ | UnsignedInt ’.’ UnsignedInt

BoolConst = true | false

*Ключові слова*

KeyWords = start | read | print | for | by | while | do | if | end

*Синтаксис*

Expression = ArithmExpression | BoolExpr

BoolExpr = Expression RelOp Expression | true | false

ArithmExpression = [ Sign] Term | ArithmExpression ’+’ Term | ArithmExpression ’-’ Term

Term = Factor | Term ’\*’ Factor | Term ’/’ Factor | ‘(‘ Tern ‘^’ Factor ‘)’

Factor = Ident | Const | ’(’ ArithmExpression ’)’

*Арифметичні операції*

AddOp = ’+’| ’-’

MultOp = ’\*’ | ’/’

NeltOp = ‘^’

*Оголошення*

IdenttList = Ident {’,’ Ident}

*Інструкції*

DoSection = StatementList

StatementList = Statement {’;’ Statement }

Statement = Assign | Inp | Out | ForStatement | IfStatment

*Присвоєення*

Assign = Ident ’=’ Expression

*Введення*

Inp = read ’(’ IdenttList ’)’.

*Виведення*

Out = print ’(’ IdenttList ’)’.

*Інструкція повторення*

ForStatement = for ‘(‘ IndExpr1 ‘)’ by ‘(‘ IndExpr2 ‘)’ while ’(‘ BoolExpr ‘)’ do ‘{‘ DoBlock ‘}’

IndExpr1 = Ident ’=’ ArithmExpression1

IndExpr2 = ArithmExpression2

DoBlock = ’{’ Statement ’}’ | ’{’ StatementList ’}’

*Умовний оператор*

IfStatement = if ‘(’ Relation ‘)’ ‘{‘ DoBlock ‘}’

Relation = BoolExpr

Таблиця лексем

|  |  |  |  |
| --- | --- | --- | --- |
| Код | Приклади лексем | Токен | Неформальний опис |
| 1 | а, xyz1, data3 | іdent | ідентифікатор |
| 2 | 123, 0, 521 | іntnum | ціле без знака |
| 3 | 34.76 | realnum | дійсне без знака |
| 4 | -6.7 | realnum | дійсне зі знаком |
| 5 | true | boolval | логічне значення |
| 6 | start | keyword | символ start |
| 7 | read | keyword | символ read |
| 8 | print | keyword | символ print |
| 9 | for | keyword | символ for |
| 10 | by | keyword | символ by |
| 14 | do | keyword | символ do |
| 16 | while | keyword | символ while |
| 17 | if | keyword | символ if |
| 18 | end | keyword | символ end |
| 19 | false | boolval | символ false |
| 20 | = | assіgn\_op | символ = |
| 21 | + | add\_op | символ + |
| 22 | - | add\_op | символ - |
| 23 | \* | mult\_op | символ \* |
| 24 | / | mult\_op | символ / |
| 25 | ^ | nelt\_op | символ ^ |
| 26 | < | rel\_op | символ < |
| 27 | <= | rel\_op | символ <= |
| 28 | == | rel\_op | символ === |
| 29 | > | rel\_op | символ > |
| 30 | >= | rel\_op | символ >= |
| 31 | != | rel\_op | символ !== |
| 32 | ( | par\_op | символ ( |
| 33 | ) | par\_op | символ ) |
| 34 | { | par\_op | символ { |
| 35 | } | par\_op | символ } |
| 36 | . | punct | символ . |
| 37 | , | punct | символ , |
| 38 | : | punct | символ : |
| 39 | ; | punct | символ ; |
| 40 | \32,\t | ws | пробільні символи |
| 41 | \n, \r, \r\n, \n\r | ls | кінець рядка |

**Базовий приклад програми**

start ProgramName  
 c = 1.0 \* 9.0 + 3.0;  
 c = -c;  
 d = 3.0 ^ (5.0 / 10.0);  
 I = 1;  
 I = -I;  
 for ( I = 1 ) by ( I = I + 1) while (I < 10) do {  
 print(I);  
 if ( I != 10 ) {  
 print (I);  
 }  
 if ( I == 10 ) {  
 print (I);  
 }  
 if ( I > 10 ) {  
 print (I);  
 }  
 if ( I < 10 ) {  
 print (I);  
 }  
  
 if ( I <= 10 ) {  
 print (I);  
 }  
 if ( I >= 10 ) {  
 print (I);  
 }  
 }  
end

* + 1. **Ідентифікатори**

Синтаксис

1. Ident = Letter { Letter | Digit | \_ }

Опис

1. Першим символом ідентифікатор може бути тільки літера, наступні символи, якщо вони є, можуть бути цифрами, літерами або символом‘\_’. Довжина ідентифікатора не обмежена.

Обмеження

1. Жоден ідентифікатор не може збігатись із ключовим словом або словами true або false.
2. Елемент, який у фазi лексичного аналiзу може бути визначений як iдентифiкатор або як ключове слово, вважається ключовим словом.
3. Елемент, який у фазi лексичного аналiзу може бути визначений як iдентифiкатор або як логiчна константа, вважається логiчною константою.

Семантика

1. Ідентифікатор може позначати змінну та програму

Візуальне представлення

1. Синтаксична діаграма

Приклади

а\_, xyz\_1, data3

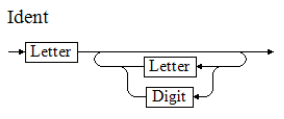


Рисунок 1 Ідентифікатор

* 1. **Константи**

Синтаксис

Const = IntNumb | RealNumb | BoolConst

IntNumb = [Sign] UnsignedInt

RealNumb = [Sign] UnsignedReal

Sign = ’-’

UnsignedInt = Digit {Digit}

UnsignedReal = ’.’ UnsignedInt| UnsignedInt ’.’ | UnsignedInt ’.’ UnsignedInt

BoolConst = true | false

Обмеження

1. Кожна константа повинна мати тип, а величина константи повинна знаходитись у діапазоні репрезентативних значень для її типу.
2. На етапі лексичного аналізу виявляються тільки беззнакові цілі константи UnsіgnedІnt, беззнакові дійсні константи UnsіgnedFloat та логічні константи BoolConst.

Семантика

1. Кожна константа має тип, визначений її формою

Візуальне представлення

1. Синтаксичні діаграми див. на рис. 1.

Приклади

1. 12, -5, 234, -1.54, 34.567, 23. , true, false

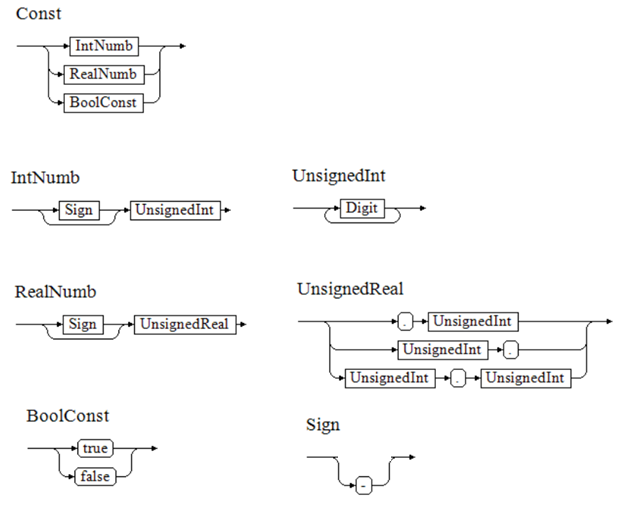


Рисунок 2 Константи

1. **Типи**

Мова KKRUT пiдтримує значення трьох типiв: integer, real та boolean.

1. Цiлий тип integer може бути представлений оголошеною змiнною типу

integer, або константою IntNumb. Дiапазон значень — [ -32768; 32767 ].

2. Дiйсний тип real може бути представлений оголошеною змiнною типу

real, або константою RealNumb. Дiапазон значень, який підтримує транслятор.

3. Логiчний тип boolean може бути представлений оголошеною змiнною типу boolean, або константою BoolConst (true або false).

Прийнято, що false < true

**Вирази**

Синтаксис

Expression = ArithmExpression | BoolExpr

BoolExpr = Expression RelOp Expression | true | false

ArithmExpression = [ Sign] Term | ArithmExpression ’+’ Term | ArithmExpression ’-’ Term

Term = Factor | Term ’\*’ Factor | Term ’/’ Factor | ‘(‘ Tern ‘^’ Factor ‘)’

Factor = Ident | Const | ’(’ ArithmExpression ’)’

Опис

1. Вираз - це послідовність операторів і операндів, що визначає порядок обчислення значення.
2. Розрізняються арифметичні та логічні вирази.
3. Значення, обчислене за арифметичним виразом, має тип float або int.
4. Значення, обчислене за логічним виразом, має тип boolean.
5. Всі бінарні оператори у виразах цієї мови лівоасоціативні, за винятком оператора ‘\*\*’ та унарного мінуса.
6. Найвищий пріоритет в унарного мінуса, далі, у порядку зменшення пріоритету слідують NeltOp, MultOp, AddOp та RelOp.
7. Послідовність двох або більше операторів з однаковим пріоритетом асоціативна.

Обмеження

1. Тип кожної змінної має бути визначений у розділі оголошень.
2. Повторне оголошеної змінної викликає помилку на етапі (у фазі) трансляції.
3. Використання неоголошеної змінної, викликає помилку на етапі трансляції.
4. Використання змінної, що не набула значення, викликає помилку.

Семантика

1. Кожна константа має тип, визначений її формою та значенням.
2. Змінна набуває значення в інструкції присвоювання Assіgn або в інструкції введення Іnp.

Візуальне представлення

1. Синтаксична діаграма.

Приклади

1. Factor:

x, 12, (a + 234)

1. Term:

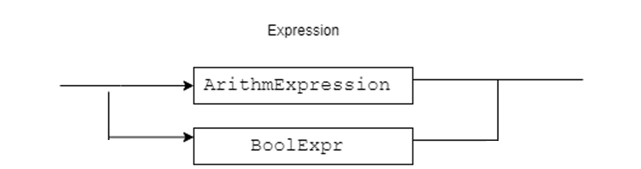
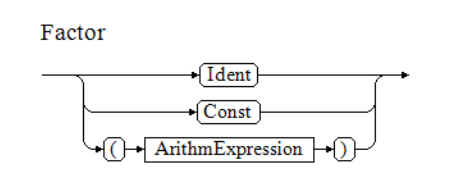
m\*z, 32/(b + 786), k\*\*n

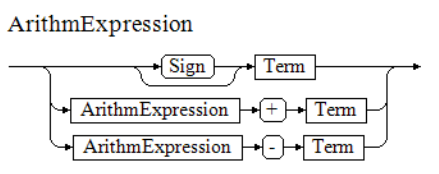
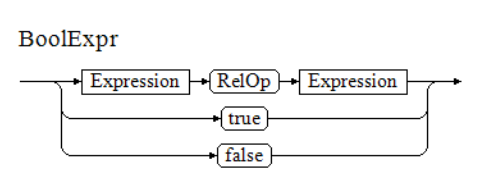
1. ArіthmExpr:

-b, f1 + g, c - 24

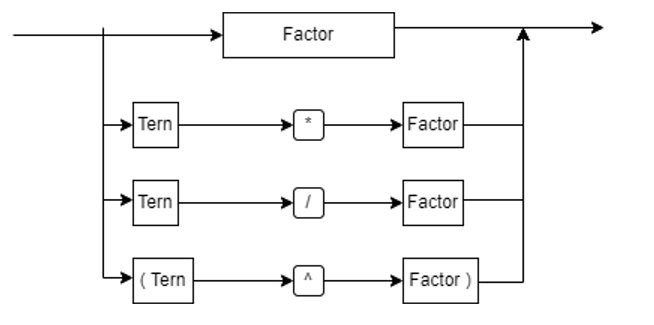
1. BoolExpr:

-b = 2, (a\*x + b/z) >= (k + t), true > false, true, false

**



Term



* + 1. **Арифметичні оператори**

Синтаксис

AddOp = ’+’ | ’-’

MultOp = ’\*’ | ’/’

NeltOp = ‘^’

Семантика

1. Тип результату при застосуванні бінарних операторів див. табл. 3.
2. Тип результату при застосуванні унарного мінуса див. табл. 4.
3. Ділення на нуль викликає помилку

Приклад

1. 0.2 \* y / 5.6, -3.4 + 3, 7 / 8, 5\*\*3.

|  |  |  |  |
| --- | --- | --- | --- |
| Оператор | Операція | Типи операндів | Тип результату |
| + | додавання | integer, real | real, якщо хоч один real, інакше - integer |
| - | віднімання |
| \* | множення |
| ^ | піднесення до степеня |
| / | ділення | real |

Таблиця 3 Бінарні арифметичні оператори

**Лексичний аналізатор**

Містить таблицю лексем, станів. Посимвольно сканує вхідний код і

перевіряє допустимість символів. Генерує таблицю символів програми,

ідентифікаторів і констант у вигляді словників. Таблиця символів містить

лексему, її токен, рядок і індекс у таблиці ідентифікаторів/констант.

Таблиця ідентифікаторів містить ідентифікатор, його тип(не визначений),

значення(не визначене) й індекс. Таблиця констант містить константу, її

тип, значення й індекс. Генерує помилку, якщо трапляється символ не з

алфавіту, неправильний формат дійсного числа або неочікуваний символ.

**Синтаксичний аналізатор**

Спрацьовує після успішного лексичного аналізу. Виконує також і

трансляцію. Зчитуючи поелементно таблицю символів, аналізує

правильність синтаксису. Генерує список для зберігання лексем коду у

постфіксній формі - польський зворотній запис.

**Таблиці**

Символiв:

tableOfSymb :

{ n\_rec : (num\_line, lexeme, token, idxIdConst) } де:

n\_rec – номер запису в таблицi символiв програми;

num\_line – номер рядка програми;

lexeme – лексема;

token – токен лексеми;

idxIdConst – iндекс iдентифiкатора або константи у таблицi

iдентифiкаторiв та констант вiдповiдно.

Ідентифiкаторiв:

tableOfId:

{ Id: (idxId, type, val) } де:

Id – iдентифiкатор (лексема);

idxId – iндекс iдентифiкатора у таблицi iдентифiкаторiв;

type – тип значення змiнної з iдентифiкатором Id, при (першому)

занесеннi до таблицi тип вважається невизначеним type\_undef;

val – значення змiнної з iдентифiкатором Id, при (першому) занесеннi

до таблицi значення вважається невизначеним val\_undef.

Констант:

tableOfConst:

{ Const: (idxConst, type, val) } де:

Const –константа (лексема);

idxConst – iндекс константи у таблицi констант;

type – тип константи; val – значення константи

**Програмна реалізація**

Функція interpret() запускає роботу інтерпретитора при умові успішного виконання синтаксичного аналізу. Поелементно зі списку постфіксного запису вилучаються лексеми. Константи і ідентифікатори кладуться у стек, реалізований класом Stack. При натрапленні на інший символ - оператор - відбувається обчислення функцією doIt().

def interpret():  
 global stack, postfixCode  
 if parseProgram():  
 maxNumb = len(postfixCode)  
 try:  
 for i in range(0, maxNumb):  
 lex, tok = postfixCode.pop(0)  
 if tok in ('integer', 'real', 'ident'):  
 stack.push((lex, tok))  
 else:  
 doIt(lex, tok)  
 if toView:  
 configToPrint(i + 1, lex, tok, maxNumb)  
 print('\nInterpretor: Інтерпретація завершена успішно')  
 return True  
 except SystemExit as e:  
 print('\nInterpreter: Аварійне завершення програми з кодом {0}'.format(e))  
 return True

Якщо прочитаний оператор - це оператор присвоювання, то виймаємо зі стека два записи - ідентифікатор і його нове значення - і змінюємо дані цього ідентифікатора у таблиці ідентифікаторів, присвоюючи нове значення і тип. Якщо прочитана лексема - це унарний мінус, то виймає один запис зі стеку і виконуємо обрахунок processingNEG() Якщо прочитана лексема - це арифметичний оператор, то виймаємо два записи і виконуємо обрахунок processingAddMultDeg()

def doIt(lex, tok):  
 global stack, postfixCode, tableOfVar, tableOfConst, tableOfLabel  
 if (lex, tok) == ('=', 'assign\_op'):  
 (lexL, tokL) = stack.pop()  
 (lexR, tokR) = stack.pop()  
 print(tokR)  
 tableOfVar[lexR] = (tableOfVar[lexR][0], tableOfConst[lexL][1], tableOfConst[lexL][2])  
 elif lex == 'NEG':  
 # зняти з вершини стека запис  
 (lexx, tokk) = stack.pop()  
  
 processingNEG((lexx, tokk))  
 elif tok in ('add\_op', 'mult\_op', 'nelt\_op'):  
 # зняти з вершини стека запис (правий операнд)  
 (lexR, tokR) = stack.pop()  
 # зняти з вершини стека запис (лівий операнд)  
 (lexL, tokL) = stack.pop()  
  
 if (tokL, tokR) in (('integer', 'real'), ('real', 'integer')):  
 failRunTime('невідповідність типів', ((lexL, tokL), lex, (lexR, tokR)))  
 else:  
 processingAddMultDeg((lexL, tokL), lex, (lexR, tokR))  
 # pass  
 return True

Обрахунок унарного мінуса. Якщо вийнятий запис - ідентифікатор, то перевіряємо, щоб він був ініціалізований, обраховуємо значення і кладемо в стек і таблицю констант.

def processingNEG(lt):  
 global stack, postfixCode, tableOfId, tableOfConst, tableOfLabel  
 lex, tok = lt  
  
 if tok == 'ident':  
 if tableOfVar[lex][1] == 'type\_undef':  
 failRunTime('неініціалізована змінна', (lex, tableOfVar[lex]))  
 else:  
 val, tok = (tableOfVar[lex][2], tableOfVar[lex][1])  
 else:  
 val = tableOfConst[lex][2]  
  
 value = -val  
 stack.push((str(value), tok))  
 toTableOfConst(value, tok)

Обрахунок арифметичної дії. Перевіряємо, щоб ідентифікатори були ініціалізовані і передаємо значення з оператором у getValue()

def processingAddMultDeg(ltL, lex, ltR):  
 global stack, postfixCode, tableOfId, tableOfConst, tableOfLabel  
 lexL, tokL = ltL  
 lexR, tokR = ltR  
  
 if tokL == 'ident':  
 if tableOfVar[lexL][1] == 'type\_undef':  
 failRunTime('неініціалізована змінна', (lexL, tableOfVar[lexL]))  
 else:  
 valL, tokL = (tableOfVar[lexL][2], tableOfVar[lexL][1])  
 else:  
 valL = tableOfConst[lexL][2]  
  
 if tokR == 'ident':  
 if tableOfVar[lexR][1] == 'type\_undef':  
 failRunTime('неініціалізована змінна', (lexR, tableOfVar[lexR]))  
 else:  
 valR, tokR = (tableOfVar[lexR][2], tableOfVar[lexR][1])  
 else:  
 valR = tableOfConst[lexR][2]  
  
 getValue((valL, lexL, tokL), lex, (valR, lexR, tokR))

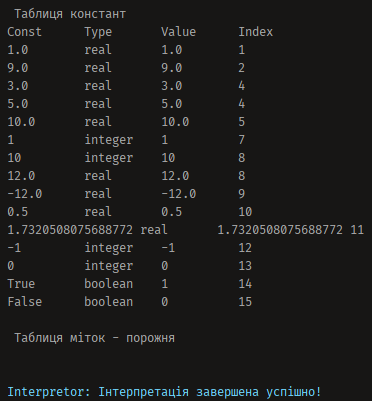
Обчислюємо нове значення відповідно до оператора. Генеруємо помилку при діленні на нуль. Кладемо нове значення на стек і у таблицю констант.

def getValue(vtL, lex, vtR):  
 global stack, postfixCode, tableOfId, tableOfConst, tableOfLabel  
 valL, lexL, tokL = vtL  
 valR, lexR, tokR = vtR  
 if lex == '+':  
 value = valL + valR  
 elif lex == '-':  
 value = valL - valR  
 elif lex == '\*':  
 value = valL \* valR  
 elif lex == '/' and valR == 0:  
 failRunTime('ділення на нуль', ((lexL, tokL), lex, (lexR, tokR)))  
 elif lex == '/':  
 value = valL / valR  
 elif lex == '^':  
 value = pow(valL, valR)  
 else:  
 pass  
 stack.push((str(value), tokL))  
 toTableOfConst(value, tokL)

**Тестування**

1. Базовий приклад:

start ProgramName  
 c = 1.0 \* 9.0 + 3.0;  
 c = -c;  
 d = 3.0 ^ (5.0 / 10.0);  
 I = 1;  
 I = -I;  
 for ( I = 1 ) by ( I = I + 1) while (I < 10) do {  
 print(I);  
 if ( I != 10 ) {  
 print (I);  
 }  
 if ( I == 10 ) {  
 print (I);  
 }  
 if ( I > 10 ) {  
 print (I);  
 }  
 if ( I < 10 ) {  
 print (I);  
 }  
  
 if ( I <= 10 ) {  
 print (I);  
 }  
 if ( I >= 10 ) {  
 print (I);  
 }  
 }  
end



Усі етапи програми були виконані та завершані успішно.

1. Помилка сканера або парсера:

* Помилка лексичного аналізу:

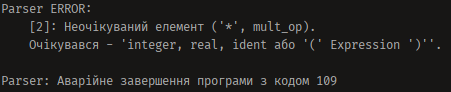
if ( I !3= 10 ) {  
 print (I);  
 }

Рисунок 22

Не правильно вказана умова порівняння.

* Помилка синтаксичного аналізу:

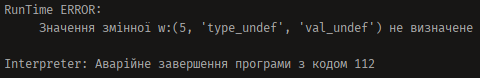
c = 1.0 \*\* 9.0 + 3.0;



При проведені арефмитичної операції була допущена помилка (\*\*, хоча повинно бути \*).

1. Невизначений один з операндів:

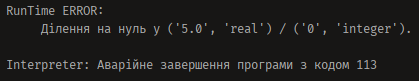
if ( w >= 10 ) {  
 print (I);  
}



При написанні коду програми не було визначено змінну g.

1. Ділення на нуль:

d = 3.0 ^ (5.0 / 0);



На нуль ділити не можна.

Висновок: в ході даної роботи я освоїв основи розробки інтерпретатору та задіяв їх на практиці. Також мною було розроблено інтерпритатор для мови програмування KK.