KAUNO TECHNOLOGIJOS UNIVERSITETAS
FAKULTETAS

Programavimo kalbų teorija (P175B124)

Projekto ataskaita

Atliko:

IFF-7/14 gr. studentai
Airidas Janonis
Eligijus Kiudys
Martynas Girdžiūna
2020 m. balandžio 30 d.

Priėmė:

Lekt. Tautvydas Fyleris
Lekt. Evaldas Guogis

KAUNAS 2020

TURINYS

1. Kalbos idėja ir pavadinimas, komandos pavadinimas ir jos nariai

Kalbos idėja: Kalba turi nesudėtingą sintaksę, kuri leidžia greitai ir efektyviai pradėti rašyti programinį kodą. Programavimo kalba pritaikyta įvairaus lygio programuotojams, tačiau labiau orientuotą į pradedančiuosius.
Pavadinimas : “S++”.
Nariai : IFF-7/14 studentai – Airidas Janonis, Eligijus Kiudys, Martynas Girdžiūna.

2. Esminės kalbos savybės

Programavimo kalba palaikys int, char ir string tipo kintamuosius. Kintamieji taip pat gali būti globalūs. Nekintančios reikšmės saugomos const tipuose.
Kalbos savybės: Nuosekliai skaitoma ir greitai suprantama programinio kodo sintaksė.

3. Baziniai ir palaikomų kalbos konstrukcijų pavyzdžiai

S++ programavimo kalba palaikys *if* sąlygos sakinį, *while* ir *for* ciklus.

Kintamųjų pavadinimai:

- int => number
- string => word
- char => letter

Kalbos konstrukcijų struktūra:

- if (condition)
- while (condition)
- for (condition)

Sisteminės funkcijos:

- print(word) - spausdinti pasirinkta tekstą į konsolę
- printLine(word) - spausdinti pasirinkta tekstą į konsolę ir perkelti žymeklį į kitą eilutę
- convertToWord(number) - pakeičia skaitinę reikšmę į tekstinę
- convertToNumber(word) - pakeičia tekstinę reikšmę (jeigu tai yra skaičius) į skaitinę reikšmę
- return value - gražinti skaitinę arba tekstinę reikšmes

Kitos struktūros:

- { } => do done

Funkcijų deklaracija:

- name (declarations (separated with semicolons)) do done

4. Unikali savybė

4.1. Savybė

Programavimo kalbos unikali savybė išsaugo visų kintamųjų buvusią reikšmę. Naudotojas turės galimybę pasiekti naujausia buvusią reikšmę su funkcijos „Previous()“ pagalbą.

4.2. kodo naudojimo pavyzdys

```
number ApskaiciuotiSkaiciu()                    //Funkcijos deklaracija
    do                                          //Pradedama funkcijos veikla
        number skaicius = 10                    //skaicius = 10
        skaicius = skaicius * 5                 //skaicius = 50
        number senasSk = skaicius.Previous()    //senasSk = 10
        senasSk = skaicius + senasSk            //senasSk = 60
        return senasSk                          //Gražinama senasSk reikšmė
    done                                        //Baigiama funkcijos veikla
```

5. Pasirinkti darbo įrankiai (ir pasirinkimo priežastys) bei darbui naudojama programavimo kalba

Buvome pasirinkę „Python“ „LLVM“ (llvmlite) kompiliatorių ir įrankių grandinių technologijų rinkinį, bet greitu metu pastebėjome, kad naudotis „Lightweight LLVM“ įrankiu butu sudėtinga, nes nėra pakankamai informacijos apie šį įrankį naudojant „Python“ programavimo kalba.
Plačiau išanalizavę interpretatoriaus kūrimo principus nusprendėme, kad naudosime „Python“ kalbą ir SLY (Sly Lex Yacc) įrankį. Šis įrankis yra PLY modernus pakaitalas. PLY implementavo „Lex“ ir „Yacc“ analizavimo įrankius „Python“ programavimo kalbai. SLY nepalaiko automatinio medžio kūrimo, tačiau suteikia visus įrankius medžio kūrimui ir interpretatoriui.

SLY įrankio šaltinis: https://sly.readthedocs.io/en/latest/index.html

6. Įrankių panaudojimas

Naudojant SLY įrankį eilutės yra paverčiamos į tokenus, kurių pagalba yra sukūriama medžio struktūra. Pagal šią struktūrą yra kuriama programavimo kalbos gramatinė sintaksė.

Naudojantis „regex“ išraiškomis gaunamas tokenas:

```
@_(r'\d+')
def NUMBER(self, t):
    t.value = int(t.value)   # Convert token to numeric
    return t
```

Pav. 1 Tokeno radimas

Iš literals sąrašo išskiriami simboliai, kurie yra naudojami norint atlikti įvairias operacijas:

```
literals = { '+', '-', '/', '*', '(', ')', '{', '}', ',', ';', '>', '<', '!', '.'}
```

Pav. 2 literals sąrašas

Remiantis gautais tokenais priskiriame „NUMBER“ tipo instrukciją:

```
@_('NUMBER')
def expr(self, p):
    return ("numberValue", int(p.NUMBER))
```

Pav. 3 NUMBER tipo instrukcija

Aprašome galutines programos instrukcijas:

```
@_('expr "+" expr',
   'expr "-" expr',
   'expr "*" expr',
   'expr "/" expr',
   'expr NE expr',
   'expr EQ expr',
   'expr ">" expr',
   'expr "<" expr',
   'expr LE expr',
   'expr ME expr',
   'expr OR expr',
   'expr AND expr')
def expr(self, p):
    return ('expr', p[1], p.expr0, p.expr1, p.lineno)
```

Pav. 4 Operaciju instrukcija

```
@_('statement')
def statements(self, p):
    return ('program', p[0], None)
```

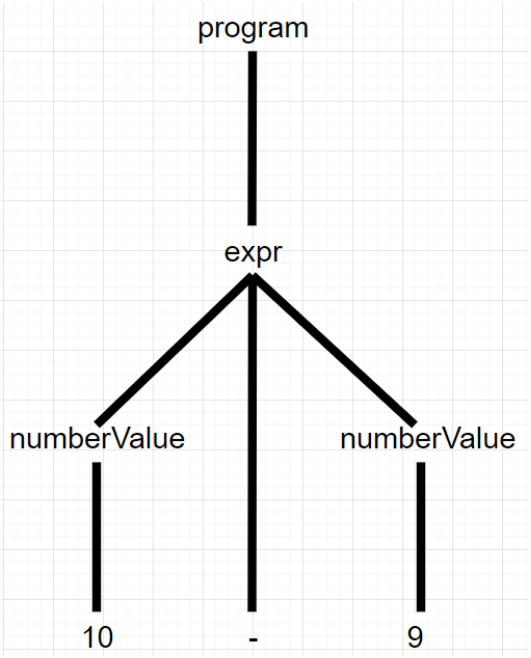
Pav. 5 Nustatoma pradinė instrukcija

Gaunama galutinė instrukciją:

```
('program', ('expr', '-', ('numberValue', 10), ('numberValue', 1), 1), None)
```

Pav. 6 Gauta galutinė instrukcija

Galutinės instrukcijos atvaizdavimas sintaksiniu medžiu:



Pav. 7 Sintaksinis medis

7. Galutinė Gramatika

```
class SppLexer(Lexer):
    # Set of token names.   This is always required
    tokens = { TYPE, ID, NUMBER, REALNUMBER, WORD, LETTER, ASSIGN,
               IF, ELSE, WHILE, DO, DONE, FOR, RETURN, STATIC,
               EQ, NE, LE, ME, OR, AND}
    ignore = '\t '
    literals = { '+', '-', '/', '*', '(', ')', '{', '}', ',', ';', '>', '<', '!', '.'}

    REALNUMBER = r'[+-]?[0-9]+\.[0-9]+'
    TYPE = r'(number)|(word)|(letter)|(real)'
    STATIC = r'static'
    IF = r'if'
    ELSE = r'else'
    WHILE = r'while'
    DONE = r'done'
    DO = r'do'
    FOR = r'for'
    RETURN = r'return'

    ID      = r'[a-zA-Z_][a-zA-Z0-9_]*'
    NUMBER  = r'\d+'
    EQ = r'=='
    ASSIGN  = r'='
    NE = r'!='
    LE = r'<='
    ME = r'>='
    OR = r'\|\|'
    AND = r'&&'


    @_(r'\".*?\"')
    def WORD(self, t):
        t.value = t.value.replace("", '')
        return t

    @_(r'\'.*?\'')
    def LETTER(self, t):
        if len(t.value) < 4:
            t.value = '{}'.format(t.value[1])
            return  t
        else:
            print('Line %d: Bad input %r' % (self.lineno, t.value))
            return None

    @_(r'\n+')
    def ignore_newline(self, t):
        self.lineno += t.value.count('\n')

    @_(r'\d+')
    def NUMBER(self, t):
        t.value = int(t.value)   # Convert token to numeric
        return t

    @_(r'#.*#')
    def COMMENT(self, t):
        pass
```

```
    @_(r'//.*')
    def COMMENT2(self, t):
        pass

    def error(self, t):
        print('Line %d: Bad character %r' % (self.lineno, t.value[0]))
        self.index += 1



class SppParser(Parser):

    tokens = SppLexer.tokens

    precedence = (
        ('left', '>', '<', LE, ME, EQ, NE, OR, AND),
        ('left', '!'),
        ('left', '+', '-'),
        ('left', '*', '/'),
        ('right', 'UMINUS'),
    )


    @_('statement statements')
    def statements(self, p):
        return ('program', p[0], p[1])

    @_('statement')
    def statements(self, p):
        return ('program', p[0], None)

    @_('statement RETURN statement')
    def statements(self, p):
        return ('program', p[0], ('program', p[2], None, p[1]))

    # statements

    # function definition

    @_('function_definition')
    def statement(self, p):
        return p.function_definition

    @_('ID "(" func_vars ")" bracket_statements')
    def function_definition(self, p):
        return ('func_def', None, p.ID, p.func_vars, p.bracket_statements)

    @_('TYPE ID "(" func_vars ")" return_bracket_statements')
    def function_definition(self, p):
        return ('func_def', p.TYPE, p.ID, p.func_vars, p.return_bracket_statements)

    @_('ID "(" ")" bracket_statements')
    def function_definition(self, p):
        return ('func_def', None, p.ID, None, p.bracket_statements)

    @_('TYPE ID "(" ")" return_bracket_statements')
    def function_definition(self, p):
```

```
    return ('func_def', p.TYPE, p.ID, None, p.return_bracket_statements)

#function definition end

# function define variables start

@_('var_declare "," func_vars')
def func_vars(self, p):
    return ("func_var", p.var_declare, p.func_vars);

@_('var_declare')
def func_vars(self, p):
    return ("func_var", p.var_declare, None);

@_('expr "," func_call_vars')
def func_call_vars(self, p):
    return ('func_call_var', p.expr, p.func_call_vars)

@_('expr')
def func_call_vars(self, p):
    return ('func_call_var', p.expr, None)

# function define variables end

# calling functions start

@_('variable_function_call')
def statement(self, p):
    return p.variable_function_call

@_('ID "." expr')
def variable_function_call(self, p):
    if p.expr[1] == "Previous":
        return ('func_call', p.expr[1], p.ID, p.lineno)
    else:
        return ('func_call', p.expr[1], ('func_call_var', ('var', p.ID, p.lineno), None), p.lineno)

# calling functions end

# loops start

@_('FOR "(" var_assign ";" expr ";" var_assign ")" bracket_statements',
   'FOR "(" var_assign ";" expr ";" var_assign ")" statement')
def statement(self, p):
    return ('for_loop', p.var_assign0, p.expr, p.var_assign1, p[8], p.lineno)

@_('WHILE "(" expr ")" bracket_statements',
   'WHILE "(" expr ")" statement')
def statement(self, p):
    return ('while_loop', p.expr, p[4], p.lineno)

# loops end

@_('if_statement')
def statement(self, p):
    return p.if_statement

    # if and else statements start
```

```
        @_('IF "(" expr ")" bracket_statements',
           'IF "(" expr ")" statement')
        def if_statement(self, p):
            return ('if_stmt', p.expr, p[4], None, p.lineno)

        @_('IF "(" expr ")" bracket_statements else_statement',
           'IF "(" expr ")" statement else_statement')
        def if_statement(self, p):
            return ('if_stmt', p.expr, p[4], p.else_statement, p.lineno)

        @_('ELSE bracket_statements',
           'ELSE statement')
        def else_statement(self, p):
            return p[1]

        @_('DO statements DONE')
        def bracket_statements(self, p):
            return p.statements

        @_( 'DO statements DONE',
            'DO RETURN expr DONE',
            'DO RETURN variable_function_call DONE',)
        def return_bracket_statements(self, p): # reikia padaryti kad eitu declare darti is naujo kvieciant
funkcija
            if(len(p) == 4):
                return ('program', p[2], None, p[1])
            else:
                return p[1]

        @_( 'var_assign',
            'var_declare',
            'expr')
        def statement(self, p):
            return p[0]

        @_('expr "+" expr',
           'expr "-" expr',
           'expr "*" expr',
           'expr "/" expr',
           'expr NE expr',
           'expr EQ expr',
           'expr ">" expr',
           'expr "<" expr',
           'expr LE expr',
           'expr ME expr',
           'expr OR expr',
           'expr AND expr')
        def expr(self, p):
            return ('expr', p[1], p.expr0, p.expr1, p.lineno)

        @_('"-" expr %prec UMINUS')
        def expr(self, p):
            return -p.expr

        @_('"!" expr')
        def expr(self, p):
            return ('not', p.expr, p.lineno)
```

```
        @_('"(" expr ")"')
        def expr(self, p):
            return p.expr

        @_('ID "(" ")" ')
        def expr(self, p):
            return ('func_call', p.ID, None, p.lineno)

        @_('ID "(" func_call_vars ")" ')
        def expr(self, p):
            return ('func_call', p.ID, p.func_call_vars, p.lineno)

        @_('NUMBER')
        def expr(self, p):
            return ("numberValue", int(p.NUMBER))

        @_('LETTER')
        def expr(self, p):
            return ("letterValue", p.LETTER)

        @_('WORD')
        def expr(sel, p):
            return ("wordValue", p.WORD)

        @_('REALNUMBER')
        def expr(sel, p):
            try:
                return ("realValue", float(p.REALNUMBER))
            except :
                pass

        @_('ID')
        def expr(self, p):
            return ('var', p.ID, p.lineno)

        @_( 'var_declare ASSIGN expr',
            'var_declare ASSIGN variable_function_call',
            'ID ASSIGN variable_function_call',
            'ID ASSIGN expr')
        def var_assign(self, p):
            return ('var_assign', p[0], p[2], p.lineno)

        @_('TYPE ID')
        def var_declare(self, p):
            return ('var_declare', p.TYPE, p.ID, False, p.lineno)

        @_('STATIC TYPE ID')
        def var_declare(self, p):
            return ('var_declare', p.TYPE, p.ID, True, p.lineno)

        def error(self, string):
            print("Error " + string.type + " at line: " + str(string.lineno) + " at index: " + str(string.index)
)
            main()
```

8. Apribojimai

- Ciklai neturi *break* išraiškos.
- Naujos kintamųjų deklaracijos tiesiog nustato juos į „default“ reikšmę, net jei jie yra funkcijoje ar cikle ir kintamasis yra globalus.
- Programa privalo būti paleista paleidus python programą ir per ją pasirinkti ar naudoti konsolinę aplinką ar naudoti failą su kodu.
- Print ir PrintLine funkcijose negalima naudoti konvertavimo funkcijų.
- Negalima naudoti dvigubų funkcijos iškvietimų kaip number.Previous().ConvertToWord()

9. Naudojimas

9.1. Paaiškinimas

Programos tekstą galima naudoti per terminalą paleidžiant *.py failą arba .exe failą* su vienu argumentu, kuris turi būti failo pavadinimas naudojant bet kokį kitą argumentą pasileidžia interpretatoriaus tekstinė sąsaja, naudojant python. paleidimą arba .exe paleidimą be argumento, pasileidžia interpretatorius kuriame per tekstinę vartotojo sąsają galima pateikti programos teksto failą, pasirinkti atskiromis eilutėmis įvedamo teksto naudojimą arba išjungti interpretatorių.

9.2. Pavyzdys

```
C:\Users\Eligijus\Desktop\KTU\Kalbu teorija\Projektas>python Spp.py main
WARNING: 11 shift/reduce conflicts
Fibonacci Recursive result 2 when number is 3
Fibonacci Recursive result 1 when number is 2
Fibonacci Recursive result 2 when number is 3
Fibonacci Loop result 2 when number is 3
Fibonacci Loop result 1 when number is 2
Fibonacci Loop result 2 when number is 3
Example function Result: 60
```

Pav. 8 Paleidžiamas interpretatorius su failu kurį iš katro paleidžia

```
C:\Users\Eligijus\Desktop\KTU\Kalbu teorija\Projektas>python Spp.py PrintLine("b")
WARNING: 11 shift/reduce conflicts
Input file (F), command lines (L) or quit (Q)?:
```

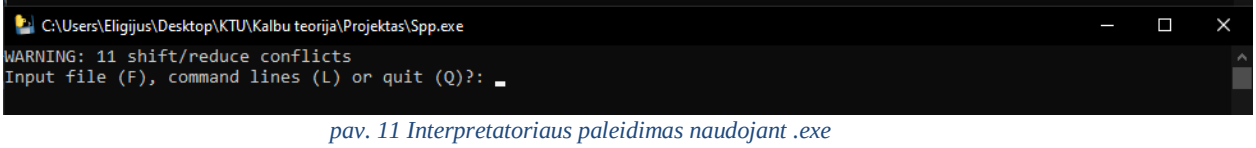
Pav. 9 Paleidžiamas interpretatorius su blogu failu

```
C:\Users\Eligijus\Desktop\KTU\Kalbu teorija\Projektas>python Spp.py
WARNING: 11 shift/reduce conflicts
Input file (F), command lines (L) or quit (Q)?:
```

Pav. 10 Paleidžiamas interpretatorius

```
C:\Users\Eligijus\Desktop\KTU\Kalbu teorija\Projektas>start Spp.exe

C:\Users\Eligijus\Desktop\KTU\Kalbu teorija\Projektas>
```



pav. 11 Interpretatoriaus paleidimas naudojant .exe

```
C:\Users\Eligijus\Desktop\KTU\Kalbu teorija\Projektas>start /b Spp.exe main

C:\Users\Eligijus\Desktop\KTU\Kalbu teorija\Projektas>WARNING: 11 shift/reduce conflicts
Fibonacci Recursive result 13 when number is 7
Fibonacci Recursive result 1 when number is 2
Fibonacci Recursive result 2 when number is 3
Fibonacci Loop result 13 when number is 7
Fibonacci Loop result 1 when number is 2
Fibonacci Loop result 2 when number is 3
Example function Result: 60
```

pav. 12 Interpretatoriaus paleidimas naudojant .exe ir failo argumentu

```
Input file (F), command lines (L) or quit (Q)?: L
s++ > PrintLine("kazkas")
kazkas
s++ >
```

Pav. 13 Komandinės eilutės naudojimo pavyzdys

```
Input file (F), command lines (L) or quit (Q)?: F
Input file: main
Fibonacci Recursive result 2 when number is 3
Fibonacci Recursive result 1 when number is 2
Fibonacci Recursive result 2 when number is 3
Fibonacci Loop result 2 when number is 3
Fibonacci Loop result 1 when number is 2
Fibonacci Loop result 2 when number is 3
Example function Result: 60
Input file (F), command lines (L) or quit (Q)?:
```

Pav. 14 Failo paleidimas interpretatoriuje

```
Input file (F), command lines (L) or quit (Q)?: Q
```

Pav. 15 Interpretatoriaus išjungimas

10.Kodas ir rezultatai

10.1.Pavyzdinis kodas

```
number n1 = 3
word nToWord1 = n1.ConvertToWord()
number n2 = 2
word nToWord2 = n2.ConvertToWord()
number n3 = 3
word nToWord3 = n3.ConvertToWord()

number recursion(number num)
do
    number ret
    if (num == 0)
    do
        ret = 0
    done
    else if (num == 1)
    do
        ret = 1
    done
    else
        ret = recursion(num - 1) + recursion(num - 2)
    return ret
done
```

```
number fibResRec = recursion(n1)
word resRecursive = fibResRec.ConvertToWord()
PrintLine("Fibonacci Recursive result " + resRecursive + " when number is " +
nToWord1)
PrintToFile("fibRec.txt", resRecursive)

number fibResRecSec = recursion(n2)
word resRecursiveSec = fibResRecSec.ConvertToWord()
PrintLine("Fibonacci Recursive result " + resRecursiveSec + " when number is "
+ nToWord2)
PrintToFile("fibRecSec.txt", fibResRecSec)

number fibResRecThird = recursion(n3)
word resRecursiveThird = fibResRecThird.ConvertToWord()
PrintLine("Fibonacci Recursive result " + resRecursiveThird + " when number is
" + nToWord3)
PrintToFile("fibRecThird.txt", fibResRecThird)

number fibonacci2(number n)
do
        number a = 0
        number b = 1
        for (number i = 0; i < n; i = i + 1)
        do
            number temp = a
            a = b
            b = temp + b
        done
        return a
done

number fibResLoop = fibonacci2(n1)
word resLoop = fibResLoop.ConvertToWord()
PrintLine("Fibonacci Loop result " + resLoop + " when number is " + nToWord1)
PrintToFile("fibLoop.txt", fibResLoop)

number fibResLoopSec = fibonacci2(n2)
word resLoopSec = fibResLoopSec.ConvertToWord()
PrintLine("Fibonacci Loop result " + resLoopSec + " when number is " +
nToWord2)
PrintToFile("fibLoopSec.txt", fibResLoopSec)

number fibResLoopThird = fibonacci2(n3)
word resLoopThird = fibResLoopThird.ConvertToWord()
PrintLine("Fibonacci Loop result " + resLoopThird + " when number is " +
nToWord3)
PrintToFile("fibLoopThird.txt", fibResLoopThird)

// naudojamas kodo pavizdys kuris buvo aprasytas pradzioje
number ApskaiciuotiSkaiciu()            //Funkcijos deklaracija
do                    //Pradedama funkcijos veikla
      number skaicius = 10            //skaicius = 10
      skaicius = skaicius * 5          //skaicius = 50
      number senasSk = skaicius.Previous()   //senasSk = 10
      senasSk = skaicius + senasSk         //senasSk = 60
      return senasSk             //Gražinama senasSk reikšmė
done

number num = ApskaiciuotiSkaiciu()
word answNum = num.ConvertToWord()
PrintToFile("exampleFunction.txt", answNum)
PrintLine("Example function Result: " + answNum)
```

10.2.Rezultatai

Konsolėje:

```
Input file (F), command lines (L) or quit (Q)?: F
Input file: main
Fibonacci Recursive result 2 when number is 3
Fibonacci Recursive result 1 when number is 2
Fibonacci Recursive result 2 when number is 3
Fibonacci Loop result 2 when number is 3
Fibonacci Loop result 1 when number is 2
Fibonacci Loop result 2 when number is 3
Example function Result: 60
```

11. Vaizdo pristatymo nuoroda

Nuoroda į vaizdo pristatyma.
https://www.youtube.com/watch?v=WkobGldZM4g