**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 3](#_Toc41170279)

[2. Esminės kalbos savybės 3](#_Toc41170280)

[3. Baziniai ir palaikomų kalbos konstrukcijų pavyzdžiai 3](#_Toc41170281)

[4. Unikali savybė 3](#_Toc41170282)

[4.1. Savybė 3](#_Toc41170283)

[4.2. kodo naudojimo pavyzdys 4](#_Toc41170284)

[5. Pasirinkti darbo įrankiai (ir pasirinkimo priežastys) bei darbui naudojama programavimo kalba 4](#_Toc41170285)

[6. Įrankių panaudojimas 5](#_Toc41170286)

[7. Galutinė Gramatika 7](#_Toc41170287)

[8. Apribojimai 12](#_Toc41170288)

[9. Naudojimas 12](#_Toc41170289)

[9.1. Paaiškinimas 12](#_Toc41170290)

[9.2. Pavyzdys 12](#_Toc41170291)

[10. Kodas ir rezultatai 13](#_Toc41170292)

[10.1. Pavyzdinis kodas 13](#_Toc41170293)

[10.2. Rezultatai 15](#_Toc41170294)

[11. Vaizdo pristatymo nuoroda 16](#_Toc41170295)

# 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.

# 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.

# 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

# Unikali savybė

## Savybė

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

## 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

# 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.

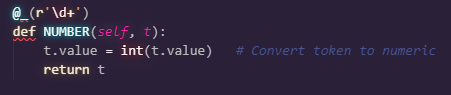
Plačiau išanalizavę interpretatoriaus kūrimo principus nusprendėme, kad naudosime „Python“ kalbą ir SLY (Sly Lex Yacc) įrankį. Šis įrankis yra PLY modernus pakaitalas. PLY implementavo „Lex“ ir „Yacc“ analizavimo įrankius „Python“ programavimo kalbai. SLY nepalaiko automatinio medžio kūrimo, tačiau suteikia visus įrankius medžio kūrimui ir interpretatoriui.

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

# 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:



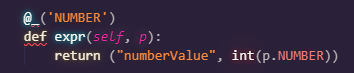
Pav. 1 Tokeno radimas

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



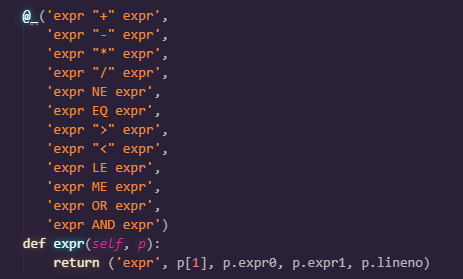
Pav. 2 literals sąrašas

Remiantis gautais tokenais priskiriame „NUMBER“ tipo instrukciją:

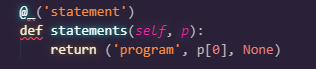


Pav. 3 NUMBER tipo instrukcija

Aprašome galutines programos instrukcijas:



Pav. 4 Operaciju instrukcija



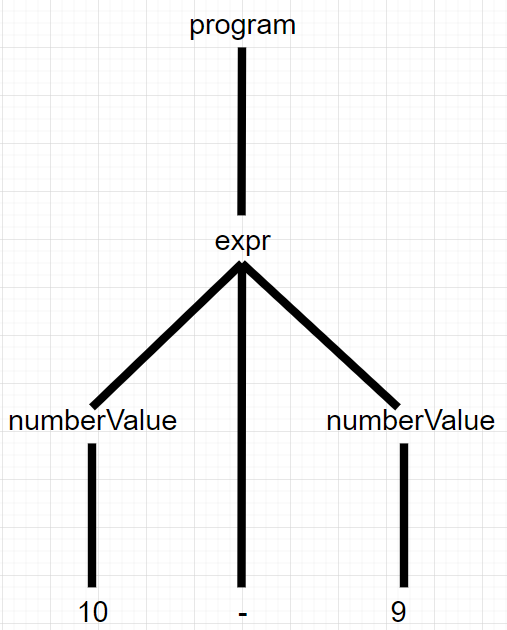
Pav. 5 Nustatoma pradinė instrukcija

Gaunama galutinė instrukciją:



Pav. 6 Gauta galutinė instrukcija

Galutinės instrukcijos atvaizdavimas sintaksiniu medžiu:



Pav. 7 Sintaksinis medis

# 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()

# Apribojimai

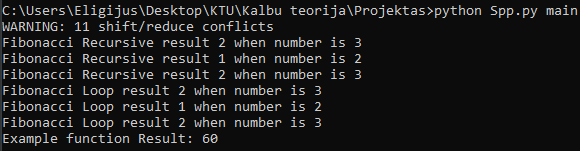
* + Ciklai neturi *break* išraiškos.
  + Naujos kintamųjų deklaracijos tiesiog nustato juos į „default“ reikšmę, net jei jie yra funkcijoje ar cikle ir kintamasis yra globalus.
* Programa privalo būti paleista paleidus python programą ir per ją pasirinkti ar naudoti konsolinę aplinką ar naudoti failą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

# Naudojimas

## 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.

## Pavyzdys



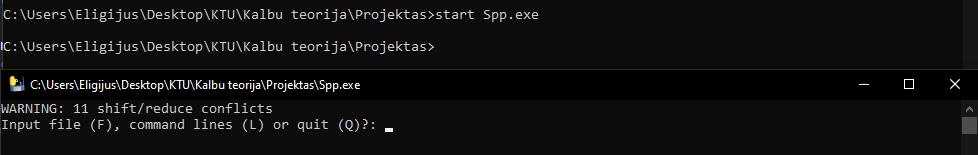
Pav. 8 Paleidžiamas interpretatorius su failu kurį iš katro paleidžia



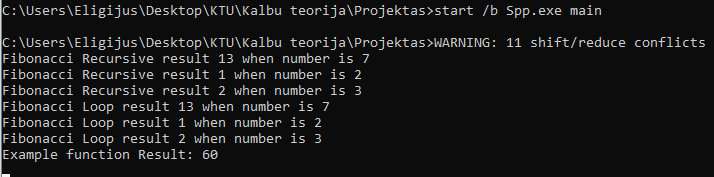
Pav. 9 Paleidžiamas interpretatorius su blogu failu



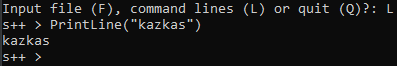
Pav. 10 Paleidžiamas interpretatorius



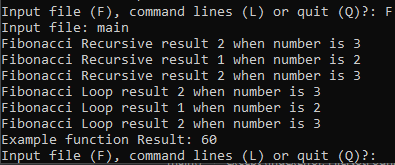
pav. 11 Interpretatoriaus paleidimas naudojant .exe



pav. 12 Interpretatoriaus paleidimas naudojant .exe ir failo argumentu



Pav. 13 Komandinės eilutės naudojimo pavyzdys



Pav. 14 Failo paleidimas interpretatoriuje



Pav. 15 Interpretatoriaus išjungimas

# Kodas ir rezultatai

## 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)

## 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

# Vaizdo pristatymo nuoroda

Nuoroda į vaizdo pristatyma.

<https://www.youtube.com/watch?v=WkobGldZM4g>