UNIVERSIDADE FEDERAL DE SANTA CATARINA

RIAN DE JESUS TURIBIO

Trabalho Final Construção de Compiladores

Araranguá

2021

RIAN DE JESUS TURIBIO

Trabalho Final Construção de Compiladores

Trabalho final da matéria apresentada ao Curso de Construção de Compiladores, da Universidade Federal de Santa Catarina.

Orientador: Prof. Alison Roberto Panisson

Araranguá

2021

SUMÁRIO

APRESENTAÇÃO INICIAL

Para o trabalho final da disciplina de construção de compiladores, foi-se pedido a construção de um compilador parcial, onde seria utilizado uma implementação anterior feita pelos alunos, a descrição do trabalho final abaixo:

Para o trabalho final da disciplina, você deve entregar uma implementação parcial de um compilador, contendo as fases de análise léxica (previamente implementada), análise sintática e análise semântica seguindo as especificações abaixo:

- Análise sintática deve ser capaz de validar comandos de atribuição, operações aritméticas simples e estruturas condicionais (simples e aninhadas)
- Análise semântica deve incluir a implementação de 3 (ou mais) ações semânticas, por exemplo, declaração de variáveis antes do uso, tipagem, e contexto.

Abaixo temos um exemplo simples de programa em C que deve ser reconhecido pelo analisador sintático.

```
int main(void)
{
   int a, b, resultado;
   a = 4;
   b = 6;
   if(a>= 10){
       resultado = a -  b;
   }else{
       resultado = a +  b;
   }
}
```

Entregáveis:

- Código fonte e arquivo README.txt descrevendo como executá-lo
- Relatório descrevendo a implementação, relacionando-a ao conteúdo teórico.

Então, a seguir irá ser demonstrado como foi implementado o seguinte trabalho.

IMPLEMENTAÇÃO

Para o inicio da implementação, a linguagem de programação python possui a possibilidade de instalação de bibliotecas para a facilitação de códigos, sem a necessidade de se criar algum framework para conseguir executar um código, para este trabalho, é utilizada a biblioteca PLY, que é um conjunto de ferramentas de análise léxica e o yacc, que é um gerador de analisador sintático, e ele gera um analisador sintático, parte do compilador responsável por fornecer sentido sintático a um determinado código fonte.

A seguir será mostrado todo código e depois dele, as explicações sobre o que as funções fazem no código:

```
# Trabalho Final - reconhecedor de estruturas em C

from ply import *

# Palavras reservadas <palavra>:<TOKEN>
reserved = {
    'if' : 'IF',
    'else' : 'ELSE',
    'for' : 'FOR',
    'while' : 'WHILE',
    'main' : 'MAIN',
    'int' : 'INT',
    'return' : 'RETURN',
    'void' : 'VOID',
}

# Demais TOKENS
tokens = [
    'EQUALS', 'GREATER', 'LESS', 'PLUS', 'MINUS', 'TIMES', 'DIVIDE', 'POWER',
    'LPAREN', 'RPAREN', 'LT', 'LE', 'GT', 'GE', 'NE',
    'GREATEQ', 'LESSEQ', 'EQEQ', 'NOTEQ', 'AND','TRUE','FALSE',
    'COMMA', 'SEMI', 'INTEGER', 'FLOAT', 'STRING','OR','DOUBLE','CHAR',
    'ID', 'NEWLINE', 'SEMICOLON', 'RBRACES', 'LBRACES'
] + list(reserved.values())

t_ignore = ' \t'

def t_REM(t):
    r'REM .*'
    return t

# Definição de Identificador com expressão regular r'<expressão>'
def t_ID(t):
    r'[a-zA-Z][a-zA-Z0-9]*'
    t.type = reserved.get(t.value,'ID')    # Check for reserved words
    return t

```

```
38 t_EQUALS = r'='
39 t_PLUS = r'\+'
40 t_MINUS = r'-'
41 t_TIMES = r'\*'
42 t_POWER = r'\^'
43 t_DIVIDE = r'/'
44 t_LPAREN = r'\('
45 t_RPAREN = r'\)'
46 t_RBRACES = r'\}'
47 t_LBRACES = r'\{'
48 t_SEMICOLON = r'\;'
49 t_LT = r'<'
50 t_LE = r'<='
51 t_GT = r'>'
52 t_GE = r'>='
53 t_NE = r'!='
54 t_COMMA = r'\,'
55 t_SEMI = r';'
56 t_INTEGER = r'\d+'
57 t_FLOAT = r'((\d*\.\d+)(E[\+-]?\d+)?|([1-9]\d*E[\+-]?\d+))'
58 t_STRING = r'\".*?\"'
59
60 def t_NEWLINE(t):
61     r'\n'
62     t.lexer.lineno += 1
63     return t
64
65 def t_error(t):
66     print("Illegal character %s" % t.value[0])
67     t.lexer.skip(1)
68
69 # Constroi o analisador léxico
70 lexer = lex.lex()
71
72 #precedence = (
73 #         ('left', 'PLUS', 'MINUS' ),
74 #         ('left', 'TIMES', 'DIVIDE' ),
75 #         ('right', 'UMINUS'),
76 #     )
77
78 precedence = (
79     ('left','AND','OR'),
80     ('left','GREATER','LESS', 'GREATEQ', 'LESSEQ', 'EQEQ', 'NOTEQ'),
81     ('left','PLUS','MINUS'),
82     ('left','TIMES','DIVIDE'),
83     ('right','UMINUS'),
84     )
85
86 names = {}
87
```

```
88  def p_estrutura_inicial(p):
89      "programa : INT MAIN LPAREN RPAREN LBRACES code RBRACES"
90
91  def p_code(p):
92          '''code : statement SEMICOLON
93                  | statement SEMICOLON code
94                  | statement IF expression LPAREN code RPAREN
95                  | statement IF expression LPAREN code RPAREN ELSE LBRACES code
96
97  #def p_statement(p):
98  #    '''
99  #    statement : if_statement
100 #              |  if_else_statement
101 #    '''
102 #    p[0] = p[1]
103
104 #def p_if_statement(p):
105 #    '''
106 #    statement : IF expression LPAREN basicblock RPAREN
107 #    '''
108 #    p[0] = IfStatement(p[2], p[4], None, lineno=p.lineno(1))
109
110 #def p_if_else_statement(p):
111 #    '''
112 #    statement : IF expression LPAREN code RPAREN ELSE LBRACES code RBRACES
113 #    '''
114 #    p[0] = IfStatement(p[2], p[4], p[8], lineno=p.lineno(1))
115
116 def p_statement_assing(p):
117         'statement : ID EQUALS expression'
118         names[p[1]] = p[3]
119
120 def p_statement_expr(p):
121         'statement : expression'
122         print(p[1])
123
124 def p_expression_binop(p):
125     '''expression : expression PLUS expression
126                   | expression MINUS expression
127                   | expression TIMES expression
128                   | expression DIVIDE expression'''
129     if p[2] == 'PLUS':
130         p[0] = p[1] + p[3]
131     elif p[2] == 'MINUS':
132         p[0] = p[1] - p[3]
133     elif p[2] == 'TIMES':
134         p[0] = p[1] * p[3]
135     elif p[2] == 'DIVIDE':
136         p[0] = p[1] / p[3]
```

```
138 def p_expression_uminus(p):
139     "expression : '-' expression %prec UMINUS"
140     p[0] = -p[2]
141 
142 def p_expression_group(p):
143     "expression : LPAREN expression RPAREN"
144     p[0] = p[2]
145 
146 def p_expression_number(p):
147     "expression : INTEGER"
148     "expression : FLOAT"
149     p[0] = p[1]
150 
151 def p_expression_double(p):
152     'expression : DOUBLE'
153     p[0] = p[1]
154 
155 def p_expression_char(p):
156     'expression : CHAR'
157     p[0] = p[1]
158 
159 def p_expression_logop(P):
160     '''expression : expression GREATER expression
161                   | expression LESS expression
162                   | expression GREATEQ expression
163                   | expression LESSEQ expression
164                   | expression EQEQ expression
165                   | expression NOTEQ expression
166                   | expression AND expression
167                   | expression OR expression'''
168     if p[2] == '>'  : p[0] = p[1] > p[3]
169     elif p[2] == '<': p[0] = p[1] < p[3]
170     elif p[2] == '>=': p[0] = p[1] >= p[3]
171     elif p[2] == '<=': p[0] = p[1] <= p[3]
172     elif p[2] == '==': p[0] = p[1] == p[3]
173     elif p[2] == '!=': p[0] = p[1] != p[3]
174     elif p[2] == '&': p[0] = p[1] and p[3]
175     elif p[2] == '|': p[0] = p[1] or p[3]
176 
177 def p_expression_name(p):
178     "expression : ID"
179     try:
180         p[0] = names[p[1]]
181     except LookupError:
182         print("Undefined name '%s'" % p[1])
183         p[0] = 0
184 
185 def p_expression_bool(p):
186     'expression : bool'
187     p[0] = p[1]
188 
```

```
188
189 def p_true(p):
190     'bool : TRUE'
191     p[0] = True
192
193 def p_false(p):
194     'bool : FALSE'
195     p[0] = False
196
197 def p_error(p):
198     if p:
199         print("Syntax error in input at token '%s'" % p.value)
200     else:
201         print("EOF","Syntax error. No more input.")
202
203 #import ply.yacc as yacc
204 #yacc.yacc()
205
206 # string de teste
207 data = '''
208
209 int main()
210 {
211     float a;
212     int b, resultado;
213      a = 4.0;
214      b = 6;
215      if(a>= 10){
216          resultado = a -  b;
217      }else{
218          resultado = a +  b;
219      }
220 }
221
222             '''
223
224 # string de teste como entrada do analisador léxico
225 lexer.input(data)
226
227 # Tokenização
228 for tok in lexer:
229      print(tok)
230
231 import ply.yacc as yacc
232 yacc.yacc()
233
234 print(data)
235
236 ## Se comentar para os erros !!!!
237 yacc.parse(data)
238
```

Primeiramente é necessário importar a biblioteca PLY para o programa, então é importada com o comando ”from ply import * ”, assim um muitas funções que deveriam ser escritas a mãos, são simplesmente chamadas para o programa, facilitando seu uso, porque a partir daí seria somente colocar os nomes das funções e fazer a chamada das mesma na biblioteca.

No programa também existe uma sequência de tokens reservados, pois seu uso é exclusivamente para funções principais, todo o código deve fornecer uma lista de tokens que define todos os nomes para tokens possíveis que podem ser produzidos pelo lexer. Esta lista é sempre necessária e é usada para realizar uma variedade de verificações de validação. A lista de tokens também é usada pelo módulo yacc.py para identificar terminais.

Abaixo temos as palavras reservadas, e os tokens para utilizar nas funções:

```
# Palavras reservadas <palavra>:<TOKEN>
reserved = {
   'if' : 'IF',
   'else' : 'ELSE',
   'for' : 'FOR',
   'while' : 'WHILE',
   'main' : 'MAIN',
   'int' : 'INT',
   'return' : 'RETURN',
   'void' : 'VOID',
}

# Demais TOKENS
tokens = [
   'EQUALS', 'GREATER', 'LESS', 'PLUS', 'MINUS', 'TIMES', 'DIVIDE', 'POWER',
   'LPAREN', 'RPAREN', 'LT', 'LE', 'GT', 'GE', 'NE',
   'GREATEQ', 'LESSEQ', 'EQEQ', 'NOTEQ', 'AND','TRUE','FALSE',
   'COMMA', 'SEMI', 'INTEGER', 'FLOAT', 'STRING','OR','DOUBLE','CHAR',
   'ID', 'NEWLINE', 'SEMICOLON', 'RBRACES', 'LBRACES'
] + list(reserved.values())
```

Para definir as expressões regulares, que são padrões de caracteres que associam sequências de caracteres no texto. Podemos usar expressões regulares para extrair ou substituir porções de texto, bem como, modificar formato de texto ou remover caracteres inválidos, para isso, as funções que usam caracteres para simbolizar soma, subtração, já são configuradas previamente, para o PLY reconhecer os caracteres. Mais especificamente, eles podem ser representados como tuplas de um token.

```
t_EQUALS = r'='
t_PLUS = r'\+'
t_MINUS = r'-'
t_TIMES = r'\*'
t_POWER = r'\^'
```

Seguindo com o código, e construída algumas regras de expressão regular, para que a biblioteca entenda na hora de fazer a verificação do código, para fazer um verificador de linhas e de erros, utiliza-se duas funções pré programadas da biblioteca:

```
def t_NEWLINE(t):
    r'\n'
    t.lexer.lineno += 1
    return t

def t_error(t):
    print("Illegal character %s" % t.value[0])
    t.lexer.skip(1)
```

Passado todas essas funções, é construído o analisador léxico do compilador, o analisador léxico, ou scanner como também é chamado, faz a varredura do programa fonte caractere por caractere e, traduz em uma sequência de símbolos léxicos ou tokens. É nessa fase que são reconhecidas as palavras reservadas, constantes, identificadores e outras palavras que pertencem à linguagem de programação. O analisador léxico executa outras tarefas como por exemplo o tratamento de espaços, eliminação de comentários, contagem do número de linhas que o programa possui:

```
lexer = lex.lex()
```

Um problema com a técnica do especificador de precedência é que às vezes é necessário alterar a precedência de um operador em certos contextos. Por exemplo, considere um operador menos em "3 + 4 * -5". Matematicamente, o menos normalmente tem uma precedência muito alta - sendo avaliado antes da multiplicação. No entanto, neste especificador de precedência, UMINUS tem uma precedência menor do que AND. Para lidar com isso, regras de precedência podem ser fornecidas para os chamados "tokens fictícios" como este:

```
precedence = (
    ('left','AND','OR'),
    ('left','GREATER','LESS', 'GREATEQ', 'LESSEQ', 'EQEQ', 'NOTEQ'),
    ('left','PLUS','MINUS'),
    ('left','TIMES','DIVIDE'),
    ('right','UMINUS'),
    )
```

Para a parte do analisador sintático, é feita uma verificação para ver como está a linha de código, se ela está sendo sendo usada como um simples identificador de variável, ou como uma estrutura condicional, estando aninhada ou simples, e verificado como a estrutura está se comportando, tendo algumas regras pré configuradas, se ela está com um IF na frente dela, ou se nao tem nada nela.

```
def p_estrutura_inicial(p):
   "programa : INT MAIN LPAREN RPAREN LBRACES code RBRACES"

def p_code(p):
      '''code : statement SEMICOLON
            | statement SEMICOLON code
            | statement IF expression LPAREN code RPAREN
            | statement IF expression LPAREN code RPAREN ELSE LBRACES code
RBRACES'''

def p_statement_assing(p):
      'statement : ID EQUALS expression'
      names[p[1]] = p[3]

def p_statement_expr(p):
      'statement : expression'
      print(p[1])
```

E para o analisador semântico, que verifica os erros semânticos no código, que dizem respeito ao escopo dos nomes, correspondência entre declarações e uso dos nomes e compatibilidade dos tipos, em expressões e comandos. Todo o tipo de verificador semântico é feito nesta parte, como verificar se uma variável é maior que outra, ou um somatório de duas variáveis, foi inserido neste código, um verificador de variáveis booleanas.

```
def p_expression_logop(P):
   '''expression : expression GREATER expression
            | expression LESS expression
            | expression GREATEQ expression
            | expression LESSEQ expression
            | expression EQEQ expression
            | expression NOTEQ expression
            | expression AND expression
            | expression OR expression'''
   if p[2] == '>'  : p[0] = p[1] > p[3]
   elif p[2] == '<': p[0] = p[1] < p[3]
   elif p[2] == '>=': p[0] = p[1] >= p[3]
   elif p[2] == '<=': p[0] = p[1] <= p[3]
   elif p[2] == '==': p[0] = p[1] == p[3]
   elif p[2] == '!=': p[0] = p[1] != p[3]
   elif p[2] == '&': p[0] = p[1] and p[3]
   elif p[2] == '|': p[0] = p[1] or p[3]
```

```
def p_expression_binop(p):
    '''expression : expression PLUS expression
                  | expression MINUS expression
                  | expression TIMES expression
                  | expression DIVIDE expression'''
    if p[2] == 'PLUS':
        p[0] = p[1] + p[3]
    elif p[2] == 'MINUS':
        p[0] = p[1] - p[3]
    elif p[2] == 'TIMES':
        p[0] = p[1] * p[3]
    elif p[2] == 'DIVIDE':
        p[0] = p[1] / p[3]
```

No final das definições semânticas, é colocado um verificador pra ver se o conteúdo do token está correto de acordo com as regras estabelecidas acima.

```
def p_error(p):
    if p:
        print("Syntax error in input at token '%s'" % p.value)
    else:
        print("EOF","Syntax error. No more input.")
```

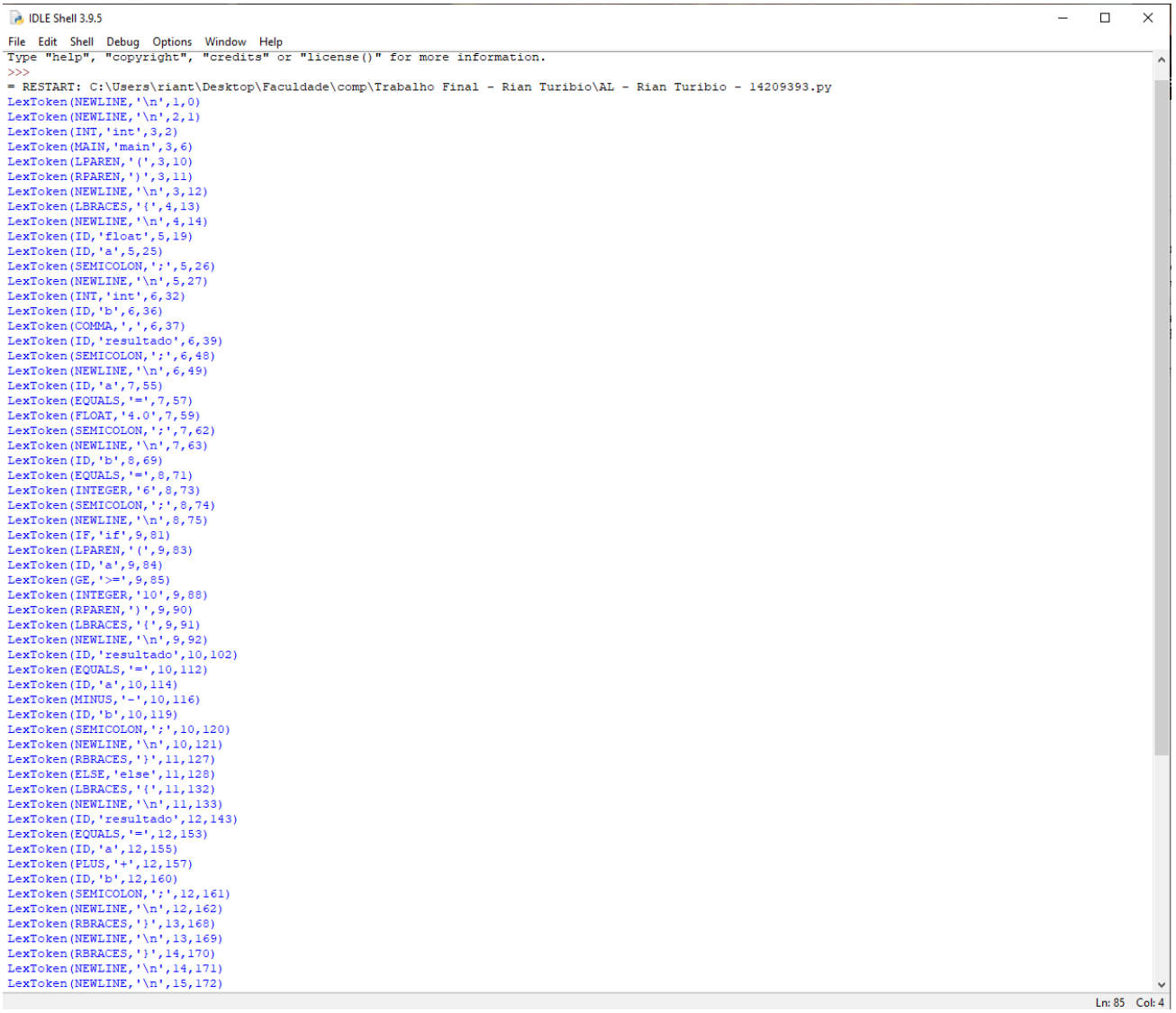
Para fazer os testes, é dado um espaço para colocar um código em linguagem C, esse código é armazenado na variável data, e a mesma é chamada em todas as partes deste compilador.

```
# string de teste
data = '''

int main()
{
    float a;
    int b, resultado;
     a = 4.0;
     b = 6;
     if(a>= 10){
        resultado = a -  b;
     }else{
        resultado = a +  b;
     }
}

          '''
```

Depois é configurada como entrada desta string de dados no analisador léxico com a função lexer.input(data), e é impresso na tela todos os resultados, e erros.

RESULTADOS

Na resultado da união de todas as partes, é impresso na tela o analisador léxico, identificando os tipos de variáveis presentes no código em C, todas os caracteres especiais, e os valores numéricos, como suas saídas, que pode ser inteiro ou em float, se o código em C estiver certo, e as definições configuradas corretamente, o código em C e impresso também na tela.



```
int main()
{
    float a;
    int b, resultado;
     a = 4.0;
     b = 6;
     if(a>= 10){
         resultado = a -  b;
     }else{
         resultado = a +  b;
     }
}
```

```
Syntax error in input at token '
'
Syntax error in input at token '
'
>>>
```