UNIVERSIDADE FEDERAL DA FRONTEIRA SUL
CAMPUS CHAPECÓ
CIÊNCIA DA COMPUTAÇÃO

CCR: CONSTRUÇÃO DE COMPILADORES

CLEISSON VIEIRA RAIMUNDI
OTÁVIO AUGUSTO DELLAY SECCO
VINICIUS DOS REIS

PROJETO E IMPLEMENTAÇÃO DAS ETAPAS DE COMPILAÇÃO PARA UMA LINGUAGEM DE PROGRAMAÇÃO HIPOTÉTICA

CHAPECÓ
2019

Resumo

Neste trabalho o objetivo foi mostrar como funciona todas as etapas de compilação e a implementação das etapas de análise léxica e sintática. Para essa implementação foi utilizado arquivos de textos que contém tokens e gramáticas regulares de uma linguagem hipotética, arquivo que contém uma gramática livre de contexto para ser utilizada pelo Gold Parser para gerar uma tabela de parsing, um arquivo .xml gerado pelo Gold Parser que é a tabela de parsing e a implementação em si dos analisadores feita em python.

Introdução

Um compilador é um programa de sistema que traduz um programa descrito em uma linguagem de alto nível para um programa equivalente em código de máquina para um processador. O compilador consiste em analisar cada símbolo do código fonte se o mesmo é permitido pela linguagem, após isso verifica se as estruturas são válidas e se existem incoerências semânticas. Se nenhum erro for encontrado o código será transformado em operações para ser otimizado.

O autômato finito implementado no componente curricular de Linguagens Formais e Autômato foi utilizado para realizar a etapa léxica do compilador junto com o Gold Parser, uma ferramenta do windows, onde dada uma Gramática Livre de Contexto irá fornecer um arquivo em formato de xml contendo a tabela LALR responsável pela etapa sintática do compilador.

Etapas de Compilação para uma Linguagem de Programação Hipotética

Um compilador é um programa de sistema que traduz um programa descrito em uma linguagem de alto nível para um programa equivalente em código de máquina para um processador. O compilador consiste em analisar cada símbolo do código fonte se o mesmo é permitido pela linguagem, após isso verifica se as estruturas são válidas e se existem incoerências semânticas. Se nenhum erro for encontrado o código será transformado em operações para ser otimizado.

Analisador Léxico

A análise léxica (ou scanner) é a primeira etapa de um compilador. Que têm a função de ler cada caractere do código fonte para identificar “elementos léxicos”: identificadores, palavras reservadas, constantes, operadores, entre outros. Também conhecidos como símbolos léxicos ou tokens. E também ignorar comentários, espaços em branco, tabs, etc.

Por exemplo, temos o seguinte código fonte: if(xis == ypis010). E uma gramática que não permite gerar qualquer token com números. O analisador léxico vai ler e separar em tokens: ‘if’, ‘(‘, ‘xis’, ‘==’, ‘ypis010’ e ‘)’. Porém como a gramática não permite tokens com número, na etapa léxica vai ser gerado um erro léxico informando a linha e o token que está errado para o usuário que compilar. Sendo assim: erro na linha 1 token ‘ypis010’.

Analisador Sintático

Se o código do usuário passar pela etapa léxica sem erros, então vai para a etapa sintática.

A análise sintática é responsável por, através da análise léxica, identificar uma sequência de tokens (elementos léxicos) e informar se é possível ser gerada a partir de uma gramática. É responsável pela estrutura/construção da linguagem.

Por exemplo, a gramática de quem construiu o compilador, ela aceita: ‘if{y == x}’. O programador digitou if(y = x). Nesse caso aconteceu um erro sintático, pois a estrutura após o ‘if’ deveria estar entre ‘()’ e não ‘{}’, o símbolo de comparação deveria ser ‘==’ e não ‘=’. É um erro de estrutura, montou essa linguagem disse que o if deveria ser de uma forma, então o programador deve obedecer essa estrutura.

Analisador Semântico

As análises léxica e sintática não estão preocupadas com o significado ou semântica dos programas que elas processam. A análise semântica tem por objetivo verificar se as construções identificadas pela análise sintática estão em acordo com as “regras semânticas” da linguagem sendo compilada, ela verifica e aponta as expressões que quebram qualquer regra determinada pela gramática. O papel do analisador semântico é prover métodos pelos quais as estruturas construídas pelo analisador sintático possam ser avaliadas ou executadas.

Exemplos típicos de erros semânticos são:

- Uma variável não declarada
- Uma multiplicação entre tipos de dados diferentes
- Atribuição de um literal para outro tipo, como um inteiro em uma string ou vice-versa.

Implementação

Foi utilizada a linguagem de programação Python devido a facilidade de manipular e disponibilidade com diferentes estruturas de dados. O git e github para controle de versão. A construção do compilador foi metódica de acordo com os conteúdos apresentados no componente curricular: ajustes no autômato finito determinístico desenvolvido no componente curricular de Linguagens Formais e Autômato assim como nos tokens e gramáticas regulares permitidos, criado um arquivo.txt que representa o código escrito pelo programador, a implementado o analisador léxico, definida a gramática livre de contexto que define as estruturas permitidas na linguagem, instalado Gold Parser no windows e usando a GLC anterior para a tabela de parsing (LR), construído o mapeamento entre o autômato finito e o Gold Parser e implementado o analisador sintático utilizando a tabela de parsing.

Arquivos:

entrada.txt informa o que é possível gerar através da gramática. De forma léxica.

config > entrada.txt

```
se
ou_se
enquanto
escreve
na_tela
novov
quebra
=
==
!=
;
{
}
(
)
/
-
+
*
,
+ior
-ior

<S> ::= .<A> | 0<C>
<A> ::= a<A> | b<A> | c<A> | d<A> | e<A> | f<A> | g<A> | h<A> | i<A> | j<A> | k<A> | l<A> | m<A> |
        n<A> | o<A> | p<A> | q<A> | r<A> | s<A> | t<A> | u<A> | v<A> | w<A> | x<A> | y<A> | z<A> | ~<B>
<B> ::= $
<C> ::= 1<C> | 2<C> | 3<C> | 4<C> | 5<C> | 6<C> | 7<C> | 8<C> | 9<C> | 0<C> | $
```

GLC.txt criada para utilizar no Gold Parser e gerar a tabela de parsing para a análise sintática.

```
"Start Symbol" = <S>
<S> ::= 'novov' <VAR0>
<VAR0> ::= '.name.' '=' '0constant' <VAR0> | <OP>
<OP> ::= <LOGIC> <OP> | <PRINT> <OP> | <SCANF> <OP> | <>
<LOGIC> ::= 'se' '(' <OPEX> ')' '{' <EXP> '}' <ELSE> | 'enquanto' '(' <OPEX> ',' <INC> ')' '{' <EXP> '}'
<ELSE> ::= 'ou_se' '(' <OPEX> ')' '{' <EXP> '}' | <>
<SCANF> ::= 'escreve' '(' '.name.' ')' ':' '0constant'
<PRINT> ::= 'na_tela' '(' '.name.' ')' | 'na_tela' '(' '0constant' ')'
<OPEX> ::= <VAR> | <CONSTANT>
<VAR> ::= '.name.' '==' '.name.' | '.name.' '!=' '.name.' | '.name.' '+ior' '.name.' | '.name.' '-ior' '.name.'
<CONSTANT> ::= '.name.' '==' '0constant' | '.name.' '!=' '0constant' | '.name.' '+ior' '0constant' | '.name.' '-ior' '0constant'
<EXP> ::= '.name.' '=' <SIMB> '*' <SIMB> | '.name.' '=' <SIMB> '/' <SIMB> | '.name.' '=' <SIMB> '-' <SIMB> | '.name.' '=' <SIMB>
<SIMB> ::= '-' '.name.' | '.name.' | '-' '0constant' | '0constant'
<INC> ::= '.name.' '+' | '.name.' '-' | '.name.' '+' '0constant' | '.name.' '-' '0constant' | <>
```

codigo.txt implementação usando a linguagem hipotética.

codigo.txt

```
novov .dasdsad~ = 0451541
se ( .x~ == .y~ ) {
.x~ = 01568 * 02568
}
na_tela ( .y~ )

```

Análise Léxica:

Percorre o arquivo codigo.txt e forma palavras/tokens para adicionar a tabela de símbolos. Cada vez que encontrar um separador vai encontrar uma palavra e então adicionar a tabela de símbolos informando a linha e o estado que reconhece essa palavra/token:

```
def analisador_lexico_sintatico(self):
    tabela = self.pegarAutomato()
    fitaS = []
    Ts = []
    codigoFonte = list(open('codigo.txt'))
    separador = [' ', '\n', '\t']
    palavra = ''
    count = 0
    estado = 0
    for linha in codigoFonte:
        count += 1
        for caracter in linha:
            if caracter in separador and palavra:
                Ts.append({'Linha': str(count), 'Estado': str(estado), 'Rotulo': palavra.strip('\n')})
                estado = 0
                palavra = ''
            else:
                try:
                    estado = tabela[estado][caracter][0]
                except KeyError:
                    estado = -1
                if caracter != ' ':
                    palavra += caracter
```

tabela de símbolos:

```
{'Linha': '1', 'Estado': '23', 'Rotulo': 'novov'}
{'Linha': '1', 'Estado': '9', 'Rotulo': '.dasdsad~'}
{'Linha': '1', 'Estado': '16', 'Rotulo': '='}
{'Linha': '1', 'Estado': '18', 'Rotulo': '0451541'}
{'Linha': '1', 'Estado': '-1', 'Rotulo': 'x'}
{'Linha': '2', 'Estado': '26', 'Rotulo': 'se'}
{'Linha': '2', 'Estado': '5', 'Rotulo': '('}
{'Linha': '2', 'Estado': '9', 'Rotulo': '.x~'}
{'Linha': '2', 'Estado': '17', 'Rotulo': '=='}
{'Linha': '2', 'Estado': '9', 'Rotulo': '.a~'}
{'Linha': '2', 'Estado': '6', 'Rotulo': ')'}
{'Linha': '2', 'Estado': '12', 'Rotulo': '{'}
{'Linha': '3', 'Estado': '9', 'Rotulo': '.a~'}
{'Linha': '3', 'Estado': '16', 'Rotulo': '='}
{'Linha': '3', 'Estado': '18', 'Rotulo': '015646'}
{'Linha': '3', 'Estado': '7', 'Rotulo': '*'}
{'Linha': '3', 'Estado': '18', 'Rotulo': '0121561'}
{'Linha': '4', 'Estado': '13', 'Rotulo': '}'}
```

Se o estado for igual a -1 significa que deu erro léxico, que não pode ser gerada essa palavra/token a partir da gramática hipotética:

```
for erro in Ts:
    if erro['Estado'] == '-1':
        print('Erro Léxico: linha "{}", erro "{}"' .format(erro['Linha'], erro['Rotulo']))
```

Mapeamento dos estados através do Gold Parser:

O mapeamento é necessário de acordo com o arquivo .xml gerado com o Gold Parser para realizar a análise sintática com a fita de saída mapeada da análise léxica. O código vai percorrer a tabela de símbolos criada na Análise Léxica e comparar com os Symbol (tabela de símbolos do .xml), adicionando o Index (que corresponder ao estado reconhecedor no .xml) dos symbol ao 'Estado' dos elementos da tabela de símbolos:

```
xml_parser = "./config/GLC.xml"
tree = ET.parse(xml_parser)
root = tree.getroot()
for symbol in root.iter('Symbol'):
    for x in Ts:
        if x['Rotulo'] == symbol.attrib['Name']:
            x['Estado'] = symbol.attrib['Index']
        elif x['Rotulo'][0] == '.' and x['Rotulo'][-1] == '~' and symbol.attrib['Name'] == '.name.':
            x['Estado'] = symbol.attrib['Index']
        elif x['Rotulo'][0] == '0' and symbol.attrib['Name'] == '0constant':
            x['Estado'] = symbol.attrib['Index']

print("\n")
for x in Ts:
    fitaS.append(x['Estado'])
    print(x)      #DEBUG TABELA DE SÍMBOLOS
fitaS.append(str(0))
print("\n Fita de saída:", fitaS, "\n")       #DEBUG FITA DE SAÍDA
```

```
Fita de saída: ['23', '9', '16', '18', '-1', '26', '5', '9', '17', '9', '6', '12', '9', '16', '18', '7', '18', '13', '0']
```

Análise Sintática:

Nessa etapa é utilizado o arquivo GLC.txt para gerar o GLC.xml através do Gold Parser instalado no windows.

Acessando o GLC.xml é possível encontrar os dados necessários para realizar a análise sintática:

```
pilha = []
pilha.append(0)
erro = 0
Rc = 0
controle = 0
for fita in fitaS:
    while 1:
        if erro == 1 or erro == -1:
            break
        for linha in root.iter('LALRState'):
            if erro == 1 or erro == -1:
                break
            elif linha.attrib['Index'] == str(pilha[-1]):
                for coluna in linha:
                    if coluna.attrib['SymbolIndex'] == fita:

                        if coluna.attrib['Action'] == '1':
                            controle = 0
                            pilha.append(fita)
                            pilha.append(int(coluna.attrib['Value']))
                            print("\nEmpilha: ", pilha)     #DEBUG EMPILHA
                            break
```

```
                        elif coluna.attrib['Action'] == '2':
                            controle = 1
                            for prod in root.iter('Production'):
                                if prod.attrib['Index'] == coluna.attrib['Value']:
                                    Rx = 2 * int(prod.attrib['SymbolCount'])
                                    break
                            if len(pilha) <= Rx:
                                erro = 1
                                break
                            for remove in range(Rx):
                                pilha.pop()
                                print("\nRedução 1: ", pilha)     #DEBUG REDUÇÃO
                            for linhaR in root.iter('LALRState'):
                                if linhaR.attrib['Index'] == str(pilha[-1]):
                                    for colunaR in linhaR:
                                        if colunaR.attrib['SymbolIndex'] == prod.attrib['NonTerminalIndex']:
                                            pilha.append(prod.attrib['NonTerminalIndex'])
                                            pilha.append(int(colunaR.attrib['Value']))
                                            Rc = 1
                                            print("\nRedução 2: ", pilha)       #DEBUG APÓS REDUÇÃO
                                            break
                                    if Rc == 1:
                                        Rc = 0
                                        break

                        elif coluna.attrib['Action'] == '4':
                            controle = 0
                            erro = -1
                            print("\nAceita: ", pilha)      #DEBUG ACEITA
                            break
            break
        if controle == 0:
            break

if erro != -1:
    print("\nErro de sintaxe!\n")
```

Conclusão

Neste trabalho foi abordado o que é um compilador, como ele funciona e as etapas para realizar a implementação do Analisador Léxico e Sintático que fazem parte do processo de um compilador. Foi dito os arquivos necessários, mostrado como eles devem ser estruturados e apresentado uma versão de implementação dos mesmos para que o leitor possa entender como funciona, realizar sua própria implementação ou até mesmo continuar a implementação citada neste artigo. Pois conforme demonstrado apenas duas etapas de compilação foram implementadas, o leitor ainda pode complementar com novas implementações ou correções e melhorias das mesmas já implementadas.

O repositório com todas as implementações está disponível em https://github.com/CleissonVieira/UFFS-CC/tree/master/Compiladores/Trab_compiladores