Construção de Compiladores: Implementação das etapas de compilação

Angemydelson Saint Bert e Bernardo Beltrame Facchi

Universidade Federal da Fronteira Sul (UFFS)
Campus Chapecó

Esse artigo descreve o desenvolvimento de uma aplicação que implementa as etapas de verificação léxica sintática de um compilador. A aplicação foi desenvolvida na linguagem Python e tem como objetivo colocar em pratica os conhecimentos adquiridos ao decorrer do semestre na disciplina de Construção de Compiladores.

1. Introdução

Esse artigo tem como objetivo descrever o desenvolvimento e funcionamento de uma aplicação que executa as etapas de verificação léxica e sintática em um compilador. As etapas citadas são feitas de acordo com arquivos contendo uma gramática regular para formação de variáveis e identificações de tokens e um arquivo contendo o código que será analisado para verificar se pertence a gramatica livre de contexto escolhida para a parte semântica.

2. Referencial Teórico

Para melhor compreensão da descrição do desenvolvimento da aplicação desse artigo se faz necessário o entendimento de alguns conceitos.

- Alfabeto: conjunto finito de símbolos ou caracteres.
- Palavra: cadeia de caracteres.
- Token: palavra reservada.
- Concatenação: junção de dois ou mais símbolos.

Um Autômato Finito é um reconhecedor de Gramaticas Regulares, gramaticas essas que constituem o grau mais simples de linguagem. Segundo Menezes, um Autômato Finito é composto por um conjunto de possíveis estados, um alfabeto, uma função de transição, um estado inicial um conjunto de estados finais [Menezes 2009].

Podemos classificar o Autômato Finito como “Determinístico” ou “Não Determinístico”.

- Autômato Finito Determinístico (AFD): Em cada um dos estados há somente uma transição para outro estado em cada um dos símbolos.
- Autômato Finito Não Determinístico (AFND): Existe pelo menos um estado em que há mais de uma transição para diferentes estados em algum símbolo.

As etapas de compilação podem ser dividas em analise léxica, analise sintática, analise semântica, geração de código intermediário e, por fim, otimização de código.

Na análise léxica, temos como objetivo identificar sequências de caracteres que constituem unidades léxicas (“tokens”). O analisador léxico lê, caractere a caractere, o

texto fonte, verificando se os caracteres lidos pertencem ao alfabeto da linguagem, identificando tokens, e desprezando comentários e espaços desnecessários [Alencar and Sirineo 2001].

Já na segunda etapa, a analise sintática, é verificado se o código está gramaticalmente correto.

Nesta etapa, é utilizado um Analisador LR para fazer a checagem sintática. Os analisadores LR (Left to right) são analisadores redutores eficientes que leem a sentença em análise da esquerda para a direita e produzem uma derivação ao mais a direita ao reverso [Alencar and Sirineo 2001].

3. Desenvolvimento

A linguagem escolhida para a implementação do projeto prático foi Python, sendo o principal motivo para a escolha a facilidade que a linguagem permite manipularmos cadeias de caracteres e listas de itens. Para manipular os arquivos XML foi usado a importação do “xml.etree.ElementTree”.

A aplicação é dividida em funções, cada uma com seu papel, indo desde gerar o AFD, até toda a lógica de redução, empilhamento, salto ou aceitação na analise sintática.

3.1. Garantindo o funcionamento do código

Existem algumas restrições e observações que devem ser levadas em conta para garantir o bom funcionamento do código.

Para garantir o bom funcionamento da geração do AFD, no arquivo “syntrax_GR.txt” devemos nos atentar a algumas regras:`

- Apenas uma gramatica ou regra por linha no arquivo fonte.
- O caractere reservado para conjunto vazio é o Epsilon.
- O estado inicial da gramatica será definido pelo “Start Symbol”.
- A letra “X” como regra de gramatica é reservada para representar o Estado de erro.
- As produções da regra devem ser separadas por uma barra vertical.
- Os estados gerados a partir de tokens são representados por números.
- O caractere reservado para conjunto vazio e o “ε”.

Já no arquivo “input_code.txt”, que contém o código que será analisado, é importante que todos os tokens e variáveis fiquem separados por um espaço em branco.

A tabela de Parser LALR deve ser gerada no software GoldParser em formato XML, e deve ficar dentro da pasta “materials”, com o nome do arquivo sendo “parser.xml”.

3.2. Criação da Tabela de ParseLALR

A criação da tabela de ParseLALR e feito com o auxílio do software GoldParser. Dada uma determinada gramatica, a aplicação em questão gera a tabela de parser, uma vez que a tabela foi gerada podemos exporta-la para um arquivo XML.

3.3. Reconhecimento léxico e sintático

O analisador léxico e sintático é implementado através da classe “Analise”.
A função “scanner” é responsável pela geração da tabela de símbolos.
A função Parser realiza efetivamente o reconhecimento sintático do programa.

Figura 1. Analisador léxico e sintático

```
# Analisadores léxico e sintático
class Analise(Inuteis):
    def __init__(self, automato):
        super(Analise, self).__init__(automato)

    def compiler(self):

        from scanner_dependencies import token_delimiters, keyword_list, integers_number

        token_delimiters = token_delimiters()

        def get_word_type(word):...

        def scanner():...

        def parser(s_table):...

        parser(scanner())
```

Figura 2. Função Scanner

```
def scanner():
    automata = self.pegarAutomato()
    has_error = False
    symbols_table = []
    word = ''; state = 0
    delimiters = set([" ", "\n", "\t"]); custom = set(
        [" ", "if", "else", "print", "while", "-", "*", "/", "+:", "+:=", ":-", ":-=", "=", "==", "(", ")", "≠", "!", "{", "}"])
    for index, line in enumerate(list(open('../materials/input_code.txt'))):
        if line.startswith('#'):
            continue

        current_line = str(index + 1)
        column = 0
        if not line.endswith(" "):
            line = line + " "
        for char in line:
            if char in delimiters and word:
                column += 1
                flag = True if word.lstrip() in custom or word == ' ' else False
                if not flag and state not in list(self.Finais): state = -1

                if len(word) > 1:
                    word_type = get_word_type(word.strip('\n').lstrip())
                    symbols_table.append(
                        {'Line': int(current_line), 'Column': column, 'State': state, 'Label': word.strip('\n').lstrip(),
                         'Type': word_type})
                state = 0; word = ' '
            else:
                try:
                    state = automata[state][char][0]
                except KeyError:
                    state = -1
                if char: word += char

    for error in symbols_table:
        if error['State'] == -1:
            print(f"(LexicalError) ⟶ Token {error['Label']} at position: file[{error['Line']}, {error['Column']}] is not valid")
            has_error = True
    if has_error: exit()
    return symbols_table
```

Figura 3. Função Parser.

```
def parser(s_table):
    tree = ET.parse('../materials/parser.xml')
    root = tree.getroot()

    symbols = [{'Name': s.get('Name'), 'Index': s.get('Index'), 'Type': s.get('Type')} for s in root.iter('Symbol')]
    productions = [{'Index': s.get('Index'), 'SymbolCount': s.get('SymbolCount'), 'NonTerminalIndex': s.get('NonTerminalIndex')} for s in
                   root.iter('Production')]

    lalr_table = []
    for lalr_state in root.iter('LALRState'):
        lalr_table.append({})
        for lalr_action in lalr_state:
            lalr_table[int(lalr_state.get('Index'))][lalr_action.get('SymbolIndex')] = {'Action': lalr_action.get('Action'),
                                                                                        'Value': lalr_action.get('Value')}

    def table_mapping():
        for symbol in symbols:
            symbol_name = symbol['Name']
            symbol_state = symbol['Index']
            for x in s_table:
                label_name = x['Label'].lstrip()
                if label_name == symbol_name:
                    x['State'] = symbol_state
                elif len(label_name) > 0 and label_name[0] == '_' and symbol_name == '_ID':
                    x['State'] = symbol_state

        s_table.append({"Line": "EOF", "Column": "EOC", "State": "0", "Label": "$", "Type": "EOF"})
        fita = [int(symb_['State']) for symb_ in s_table]
        return (s_table, fita)

    table, ribbon = table_mapping()

    stack = [0]
    while True:
        try:
            action = lalr_table[int(stack[0])][str(ribbon[0])]
        except KeyError as e:
            # retornar infos do token que ocasionou o erro sintático
            error = {"line": '', "label": ''}
            for index, tab in enumerate(table):
                if str(tab['State']) == str(e.args[0]):
                    if tab['Label'] == '$': tab = table[index - 1]
                    error.update({"line": tab['Line'], "column": tab['Column'], "label": tab['Label']})
                    break
            try:
                print(
                    f"(SyntaxError) --> Token {error['label'].lstrip()} at position: file[{error['line']}, {error['column']}] is not valid")
            except KeyError as e:
                print(f"Erro no código principal --> Verifique o arquivo de entrada se contiver espaço no final")
            break

        current_action = int(action['Action'])
        if current_action == 1:  # Empilha
            stack.insert( _index 0, ribbon[0])
            stack.insert( _index 0, action['Value'])
            ribbon.pop(0)

        elif current_action == 2:  # Reduz
            prod = productions[int(action['Value'])]
            countSymbol = int(prod['SymbolCount']) * 2
            for i in range(countSymbol): stack.pop(0)

            stack.insert( _index 0, prod['NonTerminalIndex'])
            goto = lalr_table[int(stack[1])][stack[0]]['Value']
            stack.insert( _index 0, goto)

        elif current_action == 3:  # Salto
            print('Action 3')
            exit()
            pass

        elif current_action == 4:  # Aceita
            print('OK -> Accepted')
            return table
```

3.5. Reconhecimento Sintático

Com o arquivo XML em mãos, a função Parser é responsável por percorre-lo coletando as informações importantes para a tabela, como os símbolos contidos, as

produções da gramatica com suas numerações e também os estados da tabela junto de suas ações.

Com a Fita de Saída gerada e a tabela de ParseLALR em memória, podemos executar o algoritmo de mapeamento da tabela para reconhecimento sintático.

Partindo do estado 0, seguimos a lógica do algoritmo de mapeamento da tabela, seguindo as ações indicadas na mesma (Empilhamento, Redução, Salto ou Aceitação). Em caso de erro sintético, é indicado a linha e o símbolo envolvido na ocorrência.

4. Testes e Resultados

Os testes foram feitos com a seguinte gramatica:

```
"Start Symbol" = <S>
<S> ::= <ID> '=' <EXPR> | <CONDITIONAL> | <LOOP> | 'print' '(' <ID> ')'
<CONDITIONAL> ::= 'if' <EXPR> '{' <S> '}' <CONDITIONALP>
<CONDITIONALP> ::= 'else' '{' <S> '}' | <>
<LOOP> ::= 'while' <EXPR> '{' <S> '}'
<OP> ::= '+' | '-' | '*' | '/' | '=' | '+:' | '+:=' | ':-=' | ':-' | '≠' | '=='
<EXPR> ::= <ID> | <ID> <OP> <ID> | <ID> <OP> <NUMBER> | <NUMBER> <OP> <NUMBER> | <NUMBER> <OP> <ID> | <NUMBER>
<NUMBER> ::= 1<NUM> | 2<NUM> | 3<NUM> | 4<NUM>
<NUM> ::= <>
<ID> ::= '_ID'
```

Figura 4. Gramatica Livre de Contexto (usada para etapa sintática).

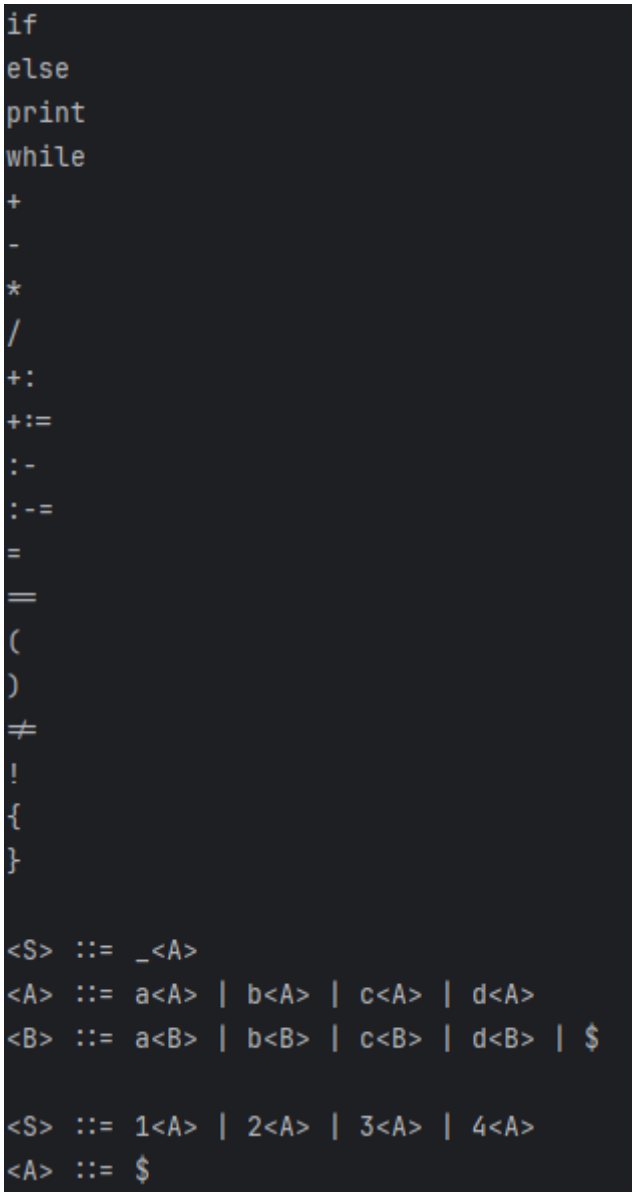

Figura 5. Gramatica Regular.

Para executarmos algum teste basta alterar o que está no arquivo “code_input.txt”, se tudo ocorrer bem ao final da execução receberemos uma mensagem informando que a sentença foi aceita.

Com o código “_a = 2 + 2” por exemplo, obtemos sucesso. Retornando “OK -> ACCEPTED”

Já com o código “_a = 2 ++ 2”, obtém-se um erro, já que o símbolo “++” não é reconhecido pela gramática.

5. Conclusão

O desenvolvimento de uma aplicação para colocar em prática o conhecimento adquirido ao longo do semestre foi um desafio que ajudou a lapidar a compreensão sobre os assuntos da disciplina, principalmente por ser necessário combinar tais conhecimentos com técnicas de programação abordadas em semestres anteriores.

Mesmo a aplicação não contemplando todas as etapas do projeto, foi possível compreender os processos por trás das etapas de compilação, suas dependências entre si e maneiras de aborda-las.

A implementação das etapas restantes fica como uma perspectiva para continuidade do trabalho.

Referenciasˆ

Alencar, P. A. M. d. and Sirineo, T. S. (2001). *Implementacao de linguagens de programacao: Compiladores*. Sagra-Luzzatto.

Menezes, P. B. (2009). *Linguagens Formais e Automatos: Volume 3 da Sˆ erie Livros´ Didaticos Inform´ atica UFRGS´* . Bookman Editora.