Ministério da Educação
Universidade Tecnológica Federal do Paraná
Campus Toledo-PR
Engenharia de Computação

EDUARDO GRZEBIELUCAS MARCHESAN
IURI SCHMOELLER
JOSÉ EDUARDO DE SOUZA

COMPILADORES

DERC

TOLEDO
2022

1. INTRODUÇÃO

A matéria de Compiladores referente ao sexto período do curso de engenharia de computação tem como objetivo geral proporcionar conhecimento teóricos e práticos com relação ao funcionamento de um compilador, assim como suas aplicações. Dessa forma são apresentados os formalismos e práticas ligados a análise léxica, sintática e semântica.

O método avaliativo da disciplina consiste na implementação de um compilador para uma linguagem de programação(LP) cujas regras, definições e etapas de projetos são definidas pelos alunos. Desse modo, fazendo com que os estudantes ponham em prática os conhecimentos obtidos na disciplina e tenham uma experiência sobre as dificuldades no projeto e implementação de um software mais complexo.

Neste documento busca-se detalhar o desenvolvimento do nosso trabalho, explicar como desenvolvemos todas as partes do compilador e justificar as decisões tomadas no projeto do compilador.

2. DESENVOLVIMENTO

O primeiro passo no desenvolvimento do nosso projeto foram as definições iniciais da linguagem que seria aceita pelo mesmo. Optamos por uma linguagem baseada majoritariamente em C, pois temos muita familiaridade e dessa forma teríamos uma maior facilidade no desenvolvimento.

Fizemos algumas mudanças na sintaxe, as quais julgamos que deixariam a linguagem mais fácil, porém a principal diferença está na definição das palavras reservadas da linguagem, que são usadas em todos os momentos quando um código é escrito e interpretado. O nome da nossa LP é “Linguagem do Bar”, onde trocamos todas as palavras mais utilizadas na programação em C para jargões utilizados em um bar. Nosso objetivo com essa escolha foi nos diferenciar dos outros grupos e chamar a atenção para nosso trabalho de forma bem humorada.

Abaixo segue uma lista das traduções da Linguagem do Bar em comparação com a linguagem C:

C	**Linguagem do Bar**
int	GARRAFA
string*	LATAO
char	LATA

float	LITRAO
if	CERVEJINHA_HOJE
else	VOU_NAO
return	ZEROU_POR_HOJE
main	BAR
while	MAIS_UMA
printf	DESCE_REDONDO
scanf	BEBER

Tabela 1 - Tradução das palavras reservadas

*string não é um formato nativo de C.

Por ser um trabalho com objetivo didático e com limitações de tempo. Podemos ver acima que nossa LP não possui todos os recursos que a linguagem C ou outras linguagens de alto nível possuem, porém tem todos os recursos necessários para resolver a maior parte dos problemas e principalmente para mostrar os conhecimentos obtidos na disciplina.

A partir da definição dos tipos de dados, e palavras reservadas, passamos para a estrutura de código. Para isso utilizamos o Formalismo de Backus-Naur (BNF) que é uma meta sintaxe usada para expressar gramáticas livres de contexto. Esse formalismo é amplamente utilizado no mundo da computação para especificação de linguagens de programação.

Durante todo o desenvolvimento a estrutura e complexidade da nossa BNF se modificou bastante. No início ela era muito simples e abrangia apenas alguns pontos da linguagem, porém conforme avançamos no desenvolvimento da análise léxica e sintática percebemos a sua real utilidade e a necessidade de complementá-la com toda a descrição da LP. Um dos fatores determinantes para isso também foram as decisões tomadas nos projetos das análises léxica e sintática que nos obrigaram a ter uma BNF bem detalhada para facilitar a implementação das regras em código.

A BNF da linguagem do bar em estágio final ficou dessa forma:

```
<s> =>                          <programa>

<programa> =>                   <sequencia_declaracao>

<empty> =>                      <>

<fim_de_instrucao> =>           PONTOVIRGULA

<literal> =>                    <numero>
                                | LITERAL

<numero> =>                     Nb_IN
                                | Nb_IN PONTO Nb_IN

<tipo> =>                       CHAR
                                | STRING
                                | FLOAT
                                | INT

<varialvel> =>                  ID
                                | ID ATRIB <expressao>

<sequencia_declaracoes> =>      <declaracao> <sequencia_declaracoes>
                                | <declaracao>

<declaracao> =>                 <funcao>
                                | <declaracao_variavel>

<declaracao_variavel> =>         <tipo> <sequencia_especificacao_var> <fim_de_instrucao>

<lista_declaracao_var> =>       <declaracao_variavel> <lista_decalracao_var>
                                | <empty> <lista_declaracao_var>

<especificacao_var > =>         ID ATRIB <expressao>
                                | ID

<sequencia_especificacao_var> => <especificacao_var> <VIRGULA> <sequencia_especificacao_var>
                                | <especificacao_var>

<parametro> =>                  <tipo> ID

<lista_parametro> =>            <sequencia_parametro>
                                | <empty> <lista_parametro>

<sequencia_parametro> =>        <parametro> <VIRGULA> <sequencia_parametro>
                                | <parametro> <sequencia_parametro>


<funcao> =>                     tipo ID ABRIRPAR <lista_paramentro> FECHARPAR ABRIRCHAVE <bloco_codigo> FECHARCHAVE

<expressao> =>                  <literal>
                                | <variavel>
                                | <expressao> OPLOGIC <expressao>
                                | ABRIRPAR <expressao> FECHARPAR
                                | <expressao> OPMAT <expressao>
                                | <subchamada_expressao> <fim_de_instrucao>

<lista_expressao> =>            <sequencia_expressao>
                                | <empty>

<sequencia_expressao> =>        <expressao> VIRGULA <sequencia_expressao>
                                |  <expressao> <sequencia_expressao>

<atribuicao> =>                 <variavel> ATRIB <expressao>

<estrutura> =>                  <condicinal>
                                | <repeticao>
                                | <retorno>
                                | <atribuicao> <fim_de_instrucao>
                                | <subchamada_expressao> <fim_de_instrucao>
                                | <escrever> <fim_de_instrucao>
                                | <ler> <fim_de_instrucao>
```

```
<lista_estrutura> =>        <estrutura> <lista_estrutura>
                            | <empty> <lista_estrutura>

<condicinal> =>             IF ABRIRPAR <expressao> FECHARPAR ABRIRCHAVE <bloco_codigo> FECHARCHAVE
                            | IF ABRIRPAR <expressao> FECHARPAR ABRIRCHAVE <bloco_codigo> FECHARCHAVE ELSE ABRIRCHAVE <bloco_codigo> FECHARCHAVE

<repeticao> =>              WHILE ABRIRPAR <expressao> FECHARPAR ABRIRCHAVE <bloco_codigo> FECHARCHAVE

<retorno> =>                RETURN <fim_de_instrucao>
                            | RETURN <expressao> <fim_de_instrucao>

<subchamada_expressao> =>   ID ABRIRPAR <lista_expressao> FECHARPAR

<escrever> =>               PRINT ABRIRPAR <lista_expressao> FECHARPAR

<ler> =>                    SCAN ABRIRPAR <lista_expressao> FECHARPAR

<bloco_codigo> =>           <lista_declaracao_var> <lista_estrutura>
```

Imagem 1 - BNF

Podemos perceber que os símbolos terminais(em caixa alta) estão somente na direita e esses podem ser traduzidos para expressões regulares correspondentes:

Símbolos Terminais	**Expressões regulares correspondentes**
PONTOVIRGULA	;
LITERAL	\"([^\\\n]|(\\.))*?\"
Nb_IN	[a-zA-Z_][a-zA-Z_0-9]*
CHAR	“LATA”
STRING	“LATAO”
FLOAT	“LITÃO”
INT	“GARRAFA”
ID	“a - z” , ”A-Z” , “a-z 0-9” , “A-Z 0-9”
ATRIB	(=)
VIRGULA	,
ABRIRPAR	(
FECHARPAR	)
ABRIRCHAVE	{

FECHARCHAVE	}
OPLOGIC	(&&) | (\#\#) | (<=) | (>=) | (==) | (<) | (>)
OPMAT	(-) | (*) | (/) | (\++)
ELSE	“VOU_NÃO”
IF	“CERVEJINHA_HOJE”
WHILE	“MAIS_UMA”
SCAN	“BEBER”
PRINT	“DESCE_REDONDO”
PONTO	.
RETURN	“ZEROU_POR_HOJE”

Tabela 2 - símbolos da linguagem

A partir de toda a estrutura da linguagem já definida, pode-se passar para a implementação das 3 partes de análise, para isso decidimos utilizar a linguagem de programação Python, baseamos nossa escolha no fato de já termos certa intimidade com essa linguagem e por ser uma LP de alto nível, não precisamos nos preocupar com detalhes de baixo nível, o que nos salvou muito tempo. Aliado ao fato de que é uma linguagem muito difundida e com uma comunidade ativa que disponibiliza muitos exemplos e bibliotecas, julgamos ser a melhor escolha para nosso projeto.

ANÁLISE LÉXICA

A principal função da análise léxica é a separação e identificação do código fonte em tokens, para fazer essa separação, podemos usar duas técnicas: autômatos finitos ou reconhecedores de expressões regulares. Decidimos usar o reconhecedor de expressões regulares já que tínhamos uma BNF e conseguimos traduzi-la para expressões regulares. Dessa forma evitamos o trabalho de construir e implementar vários autômatos.

O primeiro passo foi separar todo o código fonte em tokens, para isso buscamos uma biblioteca que pudesse nos ajudar, dando uma base para podermos progredir. Achamos a biblioteca NLTK(1), sigla para Natural Language Toolkit, como o nome sugere, ela é específica para linguagem natural e não para reconhecimento de expressões

regulares, mas como ela já realizava uma “tokenização”, decidimos usá-la como base e modificá-la conforme nossas necessidades.

De início os problemas foram relacionados a separação dos tokens. Os principais erros ocorriam quando o token era uma string, um float, um literal ou possuía algum caractere especial. Ele separava cada palavra de uma string, separava também um valor float a partir do ponto e não reconhecia caracteres especiais como parênteses, chaves e outras pontuações da forma como desejávamos. Para solucionar esses problemas, fizemos várias funções que validam o token e corrigem os erros, algumas delas são: “AjustaTokensDeLiterais”, “AjustarTokensDeCaracteresEspeciais”, “AjustaTokensDeFloat” e algumas outras que são necessárias para análise léxica como remover comentários e controle de linhas que podem ser encontrados no anexo “Primeiro Analizador Léxico.rar” ou link do link do github(3) .

Com essa parte da implementação pronta, já temos um mecanismo que detecta erros léxicos e uma tabela de tokens que armazena cada token contendo o lexema, a classe, o valor e a posição de cada um. No exemplo abaixo podemos ver um código da linguagem correspondente, assim como a tabela de símbolos gerada e a saída no terminal indicando um erro léxico e a linha em que esse erro ocorre:

```
GARRAFA BAR{
GARRAFA a,b = 7.6;
LITRAO C = 9.aae;

MAIS_UMA(i < 10 ){
        DESCE_REDONDO("Hello World");
        i = i + 1;
}
ZEROU_POR_HOJE 0;
}
```

código 1 - exemplo de código da linguagem

```
 erro léxico na linha:  [3]
PS C:\Users\eduar\OneDrive\Área de Trabalho\Primeiro Analizador Léxico>
```

código 2 - saída corespondente

```
> 1: {'Lexema': 'BAR', 'Classe': 'Keyword', 'Valor': 0, 'Posicao': 1}
> 2: {'Lexema': '{', 'Classe': 'Caractere Especial', 'Valor': 0, 'Posicao': 1}
> 3: {'Lexema': 'GARRAFA', 'Classe': 'Keyword', 'Valor': 0, 'Posicao': 1}
> 4: {'Lexema': 'a', 'Classe': 'Identificador', 'Valor': 0, 'Posicao': 1}
> 5: {'Lexema': ',', 'Classe': 'Caractere Especial', 'Valor': 0, 'Posicao': 2}
> 6: {'Lexema': 'b', 'Classe': 'Identificador', 'Valor': 1, 'Posicao': 2}
> 7: {'Lexema': '=', 'Classe': 'Operador', 'Valor': 0, 'Posicao': 2}
> 8: {'Lexema': '7.6', 'Classe': 'Float', 'Valor': 7.6, 'Posicao': 2}
> 9: {'Lexema': ';', 'Classe': 'Caractere Especial', 'Valor': 0, 'Posicao': 2}
> 10: {'Lexema': 'LITRAO', 'Classe': 'Keyword', 'Valor': 0, 'Posicao': 2}
> 11: {'Lexema': 'C', 'Classe': 'Identificador', 'Valor': 2, 'Posicao': 2}
> 12: {'Lexema': '=', 'Classe': 'Operador', 'Valor': 0, 'Posicao': 2}
> 14: {'Lexema': ';', 'Classe': 'Caractere Especial', 'Valor': 0, 'Posicao': 3}
> 15: {'Lexema': 'MAIS_UMA', 'Classe': 'Keyword', 'Valor': 0, 'Posicao': 3}
> 16: {'Lexema': '(', 'Classe': 'Caractere Especial', 'Valor': 0, 'Posicao': 3}
> 17: {'Lexema': 'i', 'Classe': 'Identificador', 'Valor': 3, 'Posicao': 3}
> 18: {'Lexema': '<', 'Classe': 'Operador', 'Valor': 0, 'Posicao': 3}
> 19: {'Lexema': '10', 'Classe': 'Inteiro', 'Valor': 10, 'Posicao': 3}
> 20: {'Lexema': ')', 'Classe': 'Caractere Especial', 'Valor': 0, 'Posicao': 3}
> 21: {'Lexema': '{', 'Classe': 'Caractere Especial', 'Valor': 0, 'Posicao': 4}
> 22: {'Lexema': 'DESCE_REDONDO', 'Classe': 'Keyword', 'Valor': 0, 'Posicao': 5}
> 23: {'Lexema': '(', 'Classe': 'Caractere Especial', 'Valor': 0, 'Posicao': 5}
> 24: {'Lexema': '"Hello World"', 'Classe': 'Literal', 'Valor': 'Hello World', 'Posicao': 5}
> 25: {'Lexema': ')', 'Classe': 'Caractere Especial', 'Valor': 0, 'Posicao': 5}
> 26: {'Lexema': ';', 'Classe': 'Caractere Especial', 'Valor': 0, 'Posicao': 5}
> 27: {'Lexema': 'i', 'Classe': 'Identificador', 'Valor': 4, 'Posicao': 5}
> 28: {'Lexema': '=', 'Classe': 'Operador', 'Valor': 0, 'Posicao': 5}
> 29: {'Lexema': 'i', 'Classe': 'Identificador', 'Valor': 5, 'Posicao': 5}
> 30: {'Lexema': '+', 'Classe': 'Operador', 'Valor': 0, 'Posicao': 5}
> 31: {'Lexema': '1', 'Classe': 'Inteiro', 'Valor': 1, 'Posicao': 6}
> 32: {'Lexema': '}', 'Classe': 'Caractere Especial', 'Valor': 0, 'Posicao': 6}
> 33: {'Lexema': 'ZEROU_POR_HOJE', 'Classe': 'Keyword', 'Valor': 0, 'Posicao': 6}
> 34: {'Lexema': '0', 'Classe': 'Inteiro', 'Valor': 0, 'Posicao': 6}
> 35: {'Lexema': ';', 'Classe': 'Caractere Especial', 'Valor': 0, 'Posicao': 6}
> 36: {'Lexema': '}', 'Classe': 'Caractere Especial', 'Valor': 0, 'Posicao': 6}
  len(): 36
```

código 3 - tabela de símbolos

Com a tabela de símbolos pronta, partimos para a análise sintática. Nosso objetivo primário era construir um parser que gerasse uma árvore sintática. Tentamos algumas implementações mas sem muito sucesso recorremos ao algoritmo CYK, um analisador para gramáticas livres de contexto. Tentamos implementar nossa própria versão, mas não obtivemos o resultado esperado. Buscamos então por implementações prontas que pudéssemos modificar para suprir nossas necessidades, mas depois de muito testar e modificar esse algoritmo, decidimos buscar por outras soluções.

Com isso, resolvemos utilizar o pacote PLY, que fornece mecanismos para realizar tanto a análise léxica quanto a análise sintática.

Na parte de análise léxica, o módulo PLY.lex() foi utilizado. Ele trabalha com expressões regulares, verificando se os lexemas do código pertencem à expressão regular que define o tipo de token. De acordo com a tabela 2, foi montada a estrutura de código para realizar a análise de acordo com as expressões regulares definidas para cada classe de token. No código 4, é possível observar um exemplo da estrutura básica de funcionamento do lexer, onde cada token é representado por uma função com o nome no formato “t_nomedotoken” (como no exemplo a classe de tokens LITERAL tem o método t_LITERAL) com a sua expressão regular correspondente na estrutura da função:

```
def t_LITERAL(t):
    r'\"([^\\\n]|(\\.))*?\"'
    return t
```

Código 4 - Estrutura básica do lexer

ANÁLISE SINTÁTICA

Posteriormente utilizando os lexemas identificados como entrada do módulo PLY.Yacc é feita a análise sintática do programa. Ela verifica se os lexemas realmente formam um código estruturado de acordo com sua gramática. Dentro do código é especificado uma gramática e as ações para cada regra. O PLY usa apenas o “:” dois pontos em vez do “::=” caracteristico de BNF para descrever suas regras. Para fazer a operação, o PLY usa a gramática descrita para gerar um analisador que usa algoritmo de redução de deslocamento, chamado de LALR (Look Ahead Left-to-right), o qual utiliza uma pilha e um autômato finito. A sequência de lexemas é lida e processada da esquerda para a direita, aplicando-se as regras quando possível, até que todo o código seja reduzido a uma regra principal, que define se o programa deve ser uma sequência de um ou mais procedimentos

Cada regra de produção da BNF da linguagem corresponde a uma função no módulo Yacc, o nome da função deve seguir o padrão “p_nomedaregra”, com a string que

representa a regra no formato de BNF. No código 5 têm-se o exemplo da função correspondente a regra de produção “estrutura” presente na imagem 1:

```
def p_estrutura(p):
    '''estrutura : condicional
                 | repeticao
                 | retorno
                 | atribuicao fim_de_instrucao
                 | subchamada_expressao fim_de_instrucao
                 | escrever fim_de_instrucao
                 | ler fim_de_instrucao
                 | chamada_funcao fim_de_instrucao'''
```

Código 5 - Estrutura básica do parser

Para todas as regras de produção da linguagem, foi definida uma função correspondente, assim possibilitando a construção do analisador sintático por parte do pacote PLY. Basta apenas indicar o estado inicial de produção que o analisador é construído, para a nossa BNF o estado inicial é programa, no código 6 é possível observar o comando utilizado para inicializar o parser:

```
parser=yacc.yacc(start='programa')
```

Código 6 - Inicialização do parser

ANÁLISE SEMÂNTICA

A análise semântica utilizada foi bem sucinta e simples, devido ao tempo reduzido para o desenvolvimento do compilador. Não houve necessidade de utilizar a árvore abstrata de sintaxe, pois foi possível detectar os erros desejados por meio das funções utilizadas na análise sintática. Apenas a detecção de unicidade de variáveis e a verificação se a variável já foi declarada posteriormente foram feitas.

No código 7, é possível observar que é possível utilizar o parâmetro p para verificar os elementos presentes na regra de produção, onde p[1] é o primeiro elemento a direita da regra, correspondente a um token “ID”. É feita uma verificação na regra de atribuição, onde o token ‘ID’ utilizado deve estar na lista de nomes posteriormente declarados:

```
def p_atribuicao(p):
    '''atribuicao : ID ATRIB expressao'''
    if(p[1] not in nomes and not SemanticErrorConstants.errorThrown):
        SemanticErrorConstants.errorThrown = True
        print("Erro semântico - Variavel '%s' não declarada na linha %d" % (p[1],p.lineno(1)-error.ErrorConstants.lineNumbers))
```

Código 7 - Verificação de erro semântico

Da mesma forma, as outras regras semânticas que foram verificadas, também estão implementadas dentro das funções das regras de produção sintáticas.

3. CONCLUSÃO

As linguagens de programação são muito complexas, pois dependem de várias etapas de análise para obter seu resultado satisfatório em executar uma sequência de passos. Ao criar a linguagem BAR e começar entender as maneiras de realizar o código, nos deparamos na infinidade de possibilidades que possui o compilador e que para criar um do zero e funcional, há um enorme nível de complexidade. Contudo utilizamos de bibliotecas e métodos de linguagens modernas para facilitar esses passos e conseguirmos uma análise satisfatória. Apesar do tempo não ter sido suficiente, ainda conseguimos realizar uma boa parte, chegando a identificar se o que o usuário está programando é válido. Ficando assim a ideia para trabalhos futuros, dar sequência ao desenvolvimento dessa linguagem.

4. REFERÊNCIAS

1. NLTK TEAM. **Natural Language Toolkit**. Disponível em: https://www.nltk.org/index.html#natural-language-toolkit. Acesso em: 20 out. 2022.
2. SIEK, Jeremy G. **Assignment 2: Parsing with PLY (Python Lex-Yacc)** Disponível em: https://www.researchgate.net/publication/242373397_Assignment_2_Parsing_with_PLY_Python_Lex-Yacc.
3. MARCHESAN, Eduardo G. **Analise-Lexica-compilador**. Disponível em: https://github.com/EduMarck/Analise-Lexica-compilador. Acesso em: 12 dez. 2022.