Construção de Compiladores
Etapas de Compilação

Raphael Borges[1], Raquel Hengen Ribeiro[1]
Universidade Federal da Fronteira Sul[1]

16 de dezembro de 2019

Resumo

Este artigo tem como objetivo fazer uma revisão teórica e mostrar a realização da implementação das etapas de compilação na construção para uma linguagem de programação hipotética, atendendo os requisitos avaliativos de cada etapa. Neste trabalho foi possível fazer as implementações análise léxica e análise sintática a partir da implementação do gerador do automato finito determinístico.

1 Introdução

Este projeto tem como objetivo fazer a implementação das etapas de compilação, Análise Léxica, Análise Sintática, Análise Semântica, Geração de Código Intermediário e Otimização para uma linguagem de programação hipotética, atendendo os requisitos avaliativos de cada etapa presentes na descrição do trabalho.

2 Referencial Teórico

De acordo com as definições de Alfres V. Aho, Monica S. Lam e Ravi Sethi Jeffrey no livro Compiladores: princípio, técnicas e ferramentas "um compilador é um programa que recebe como entrada um programa em linguagem de programação - a linguagem *fonte* - e o traduz para um programa equivalente em outra linguagem - a linguagem *objeto*"(SETHI; ULLMAN; LAM, 2008), ou seja, o compilador é quem transforma o código em código de máquina, assim, fazendo a tradução entre o que foi escrito pelo programador e o que a maquina executará. Para os autores do livro "um papel importante do compilador é relatar quaisquer erros no programa fonte detectados durante esse processo de tradução"(SETHI; ULLMAN; LAM, 2008).

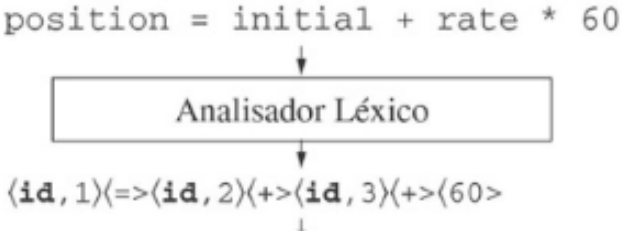

Figura 1 – Esquema de entrada e saída da análise léxica, presente no livro: Compiladores: princípios técnicas e ferramentas. (SETHI; ULLMAN; LAM, 2008)

2.1 Etapas de Compilação

Para que essa tradução possa ocorrer, como descrito no livro Compiladores: princípio, técnicas e ferramentas de Alfres V. Aho, Monica S. Lam e Ravi Sethi Jeffrey ,um compilador em duas fases a *análise* e a *síntese*, onde a *análise* "divide o programa fonte nas partes constituintes e cria uma representação intermediário do mesmo"(AHO et al., 1990) e a *síntese* "constrói o programa alvo desejado, a partir da representação intermediaria"(AHO et al., 1990) e a junção dessas partes são algumas etapas, as etapas de compilação, que veremos a seguir, que geram e utilizam da tabela de símbolos que é responsável pelo armazenamento das informações sobre todo o programa fonte para garantir a verificação e execução correta do código. As etapas de compilação são: Analise Léxica, Análise Sintática e Análise Semântica que formam a fase de análise e a Geração de Código Intermediário e Otimização de Código Dependente da Máquina que formam a fase de síntese.

2.1.1 Análise Léxica

A análise léxica é a primeira etapa dentre as etapas de compilação. Ela segundo Alfres, Monica e Ravi consite em uma analisador léxico, que lê o fluxo de caracteres que compõem o programa fonte e os agrupa em sequencia significativas, chamadas *lexemas*. Pata cada lexema, o analisador léxico produz como saída um *token* e esse *token* é passado para a próxima etapa (SETHI; ULLMAN; LAM, 2008), como podemos ver na Figura - reffig:anLexica onde podemos ver visualmente o fluxo da análise léxica

2.1.2 Análise Sintática

Na etapa seguinte, a análise sintática, o analisador sintático gera a estrutura gramatical através de uma árvore, a arvore de sintaxe que é formada pela sequencias dos *tokens* recebidos da etapa anterior, em que cada nó interior representa a operação, os filhos do nó representam os argumentos da operação.*token*, através da Figura - 2 observamos o comportamento desta etapa com a etapa anterior.

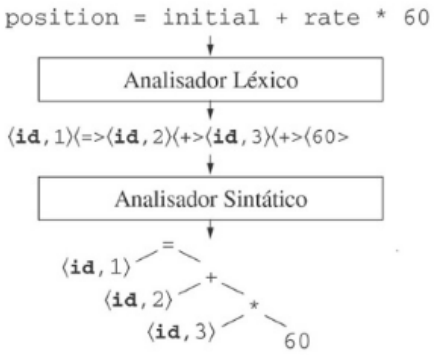

Figura 2 – Esquema de entrada e saída da análise sintática, presente no livro: Compiladores: princípios técnicas e ferramentas. (SETHI; ULLMAN; LAM, 2008)

A estruturação da gramática utilizando a árvore de sintaxe é o que vai auxiliar as análises do código e gerar o programa objeto, em outras palavras, a análise sintática verifica se lá algum erro de sintaxe no código.

2.1.3 Análise Semântica

Nesta etapa o analisador semântico utiliza da arvore de sintaxe gerada pela etapa anterior junto as informações da tabela de símbolo parar verificar semântica do programa fonte em comparação com a linguagem, em outras palavras a análise semântica faz a verificação das definições da linguagem como o programa fonte mantendo as informações relativas das declarações num base de dados (a tabela de símbolos) que é consultada durante a verificação ,assim como mostrado pela figura 3.

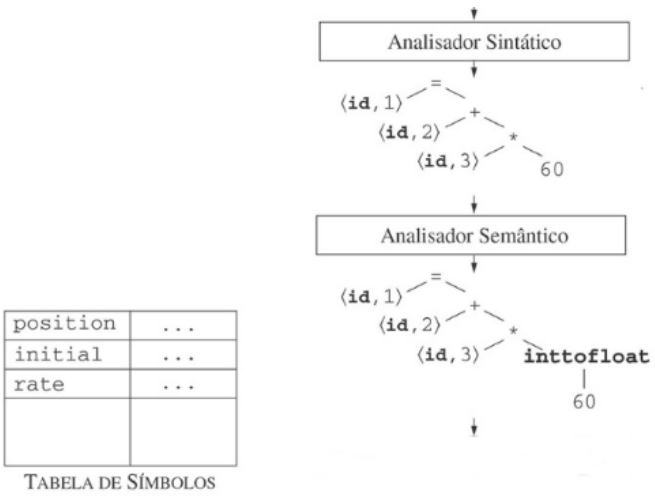

Figura 3 – Esquema de entrada e saída da análise semântica, presente no livro: Compiladores: princípios técnicas e ferramentas. (SETHI; ULLMAN; LAM, 2008)

É nesta fase também que são verificadas as tipagens de cada operador e checa sua compatibilidade, faz os processo de coerções também conhecido como *cast* ou conversão de tipagem, ex: convertendo um *int* para *float*.

2.1.3.1 Tabela de Símbolos

A tabela de símbolos ela é utilizada para armazenar os identificadores das constantes, funções, variáveis e tipos de dados presentes no programa fonte, sendo uma base de dados para consulta pelas etapas de compilação, para que toda vez que um nome seja encontrado a tabela é consultada ou alterado em caso de novas informações. Podemos ver a tabela de símbolos representada dela figura 3, presente no livro: Compiladores: princípios técnicas e ferramentas. (SETHI; ULLMAN; LAM, 2008)

2.1.4 Geração de Código Intermediário

Com o incio dessa etapa também se da início da nova fase de compilação, a fase de síntese. É nessa etapa onde se inicia o processo de tradução do programa fonte no programa objeto que é o programa que será executado.

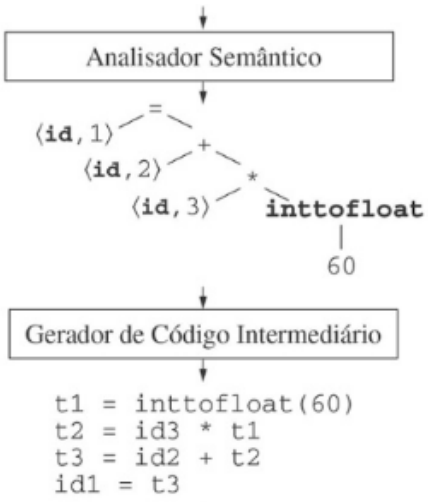

Figura 4 – Esquema de entrada e saída da Geração de Código Intermediário, presente no livro: Compiladores: princípios técnicas e ferramentas. (SETHI; ULLMAN; LAM, 2008)

Através do que é recebido pela análise semântica, será assim então, gerado do código com todas as instruções presentes no programa fonte gerando o código objeto, como representado na figura 4.

2.1.5 Otimização

Na etapa da de otimização vai pegar o código objeto gerado na etapa anterior e vai otimizar ele o máximo possível, como visto na figura 5, sendo a otimização focada

em gerar um programa mais rápido, com um código menor ou que consuma menor energia, este número de otimizações varia muito de compilador para compilador e porém quanto mais otimizações, maior o atraso na compilação do programa.

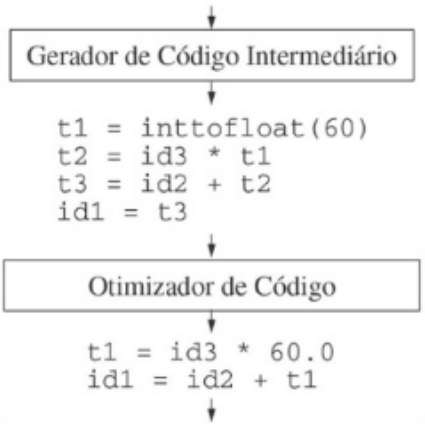

Figura 5 – Esquema de entrada e saída da Otimização do código, presente no livro: Compiladores: princípios técnicas e ferramentas. (SETHI; ULLMAN; LAM, 2008)

2.1.6 Geração de Código

Por fim com o código objeto já otimizado, ele será traduzido para a linguagem de máquina a qual a linguagem de máquina de alguma arquitetura a qual aquela linguem foi pensada para ser utilizada, conforme vemos na figura 6

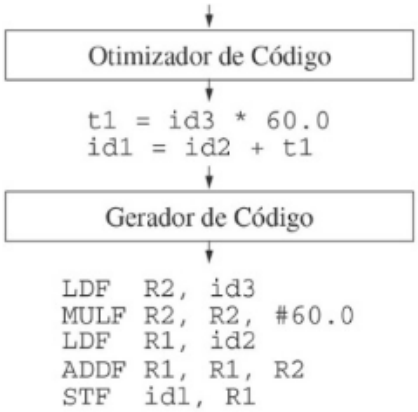

Figura 6 – Esquema de entrada e saída da Geração do Código de instrução de máquina, presente no livro: Compiladores: princípios técnicas e ferramentas. (SETHI; ULLMAN; LAM, 2008)

2.2 Compilação do Código e Erros de Compilação

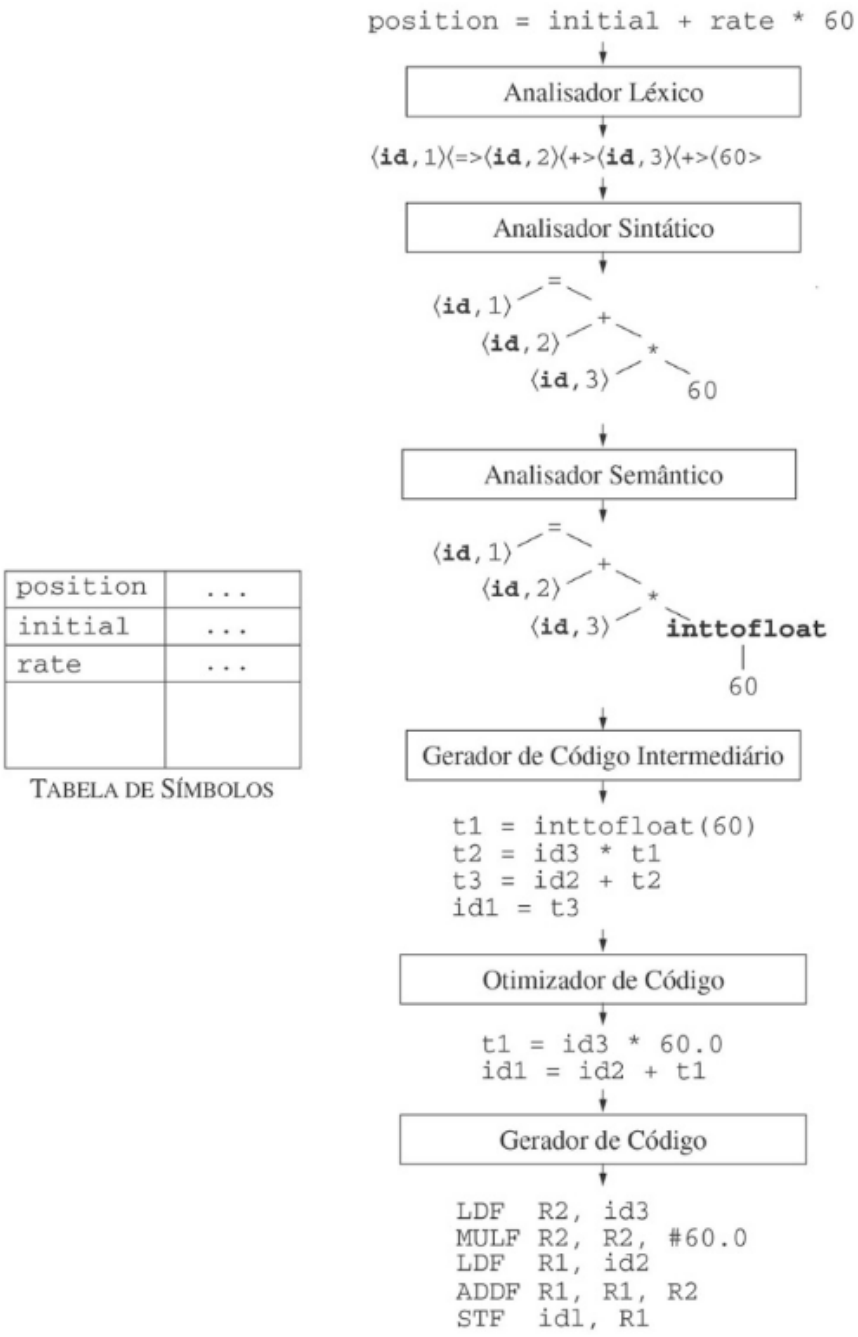

Figura 7 – Representação do fluxo de compilação, presente no livro: Compiladores: princípios técnicas e ferramentas. (SETHI; ULLMAN; LAM, 2008)

Sendo assim, como vemos na figura 7 o fluxo para a compilação de um programa fonte até a geração do código de instrução de máquina são feitas em uma seríe de etapas onde cada uma delas gera suas saídas e erros. Esses erros quando bem

informadas facilita o programador identificar onde o programa está com problemas e onde tratar.

3 Implementação e Resultados

Neste trabalho foi utilizado a linguagem de programação Python em conjunto com as bibliotecas Pandas e Numpy, sendo implementados apenas as etapas da Análise Léxica e Análise Sintática dentre as demais etapas de compilação.

3.1 Gerador do Automato Finito Determinístico

Para poder realizar as etapas de compilação utilizado um Gerador do Automato Finito Determinístico (GAFD), para a gerar o automato finito determinístico da linguagem que criamos, em que os *tokens* da linguagem são: *SELECT,WHERE, FROM, INSERT, INTO, VALUES, UPDATE, SET, DELETE, CREATE e TABLE.* e esta linguagem também permite a criação de variáveis e utilização caracteres numéricos no programa fonte.

Como implementaçaçã do GAFD utilizamos a implementação do Eduardo Stefanello feito na disciplina de Linguagens Formais e Automatas (LFA) disponível em seu GitHub[1] como ponta pé inicial para os demais processos necessários para alcançar os objetivos deste trabalho.

3.2 Implementação Análise Léxica

Na etapa de análise léxica teve como entrada o Automato Finito determinístico, suas regras e quais regras eram estados finais, e o arquivo de entrada (o código para o reconhecimento). A implementação consiste em percorrer o arquivo de entrada *token* à *token* conforme vemos no na implementação 1 verificando se tal pertence a linguagem tal como na implementação 2, quando encontra um separador ([' ',"]), ele verifica se a cadeia de caracteres pertence ao conjunto dos estados finais, se sim adiciona o *token*, a regra e a linha na tabela de símbolos.

Listing 1 – percorre codigo

```
def __percorrecodigo(self, codigo):
        for i in open(codigo).read():
            self.__testacaractere(i)
            self.coluna += 1
```

Listing 2 – percorre codigo

```
    def __testacaractere(self, i):
        if i in self.regras[self.estado]:
```

[1] https://github.com/dudustefanello/TrabalhoLFA

```
            self.token.append(i)
            self.__caracterevalido(i)
        elif i in self.separadores:
            self.__reconheceestado(i)
            self.__novalinha(i)
        else:
            raise ErroLexico(1, i, self.linha, self.coluna)
```

Como saída da análise léxica temos a tabela de símbolos, como podemos ver a seguir no código 3 uma parte dela, gerada a partir da gramatica, que será entrada fundamental para as demais etapas de compilação.

Listing 3 – Tabela de Símbolos

```
Regra, linha, token
(129,   0,      'DELETE')
(112,   0,      'FROM')
(69,    0,      'produtos')
(127,   0,      'WHERE')
(69,    0,      'codigo')
('$',   0,      '$')
(76,    0,      '=3')
```

3.3 Implementação Análise Sintática

Para construção da etapa de análise sintática foi necessário várias fases. A primeira foi a construção regras sintáticas da linguagem livre de contexto mostradas na figura 8. Os itens em maiúsculo representam produções não terminais, e os itens em minúsculo representam as produções terminais.

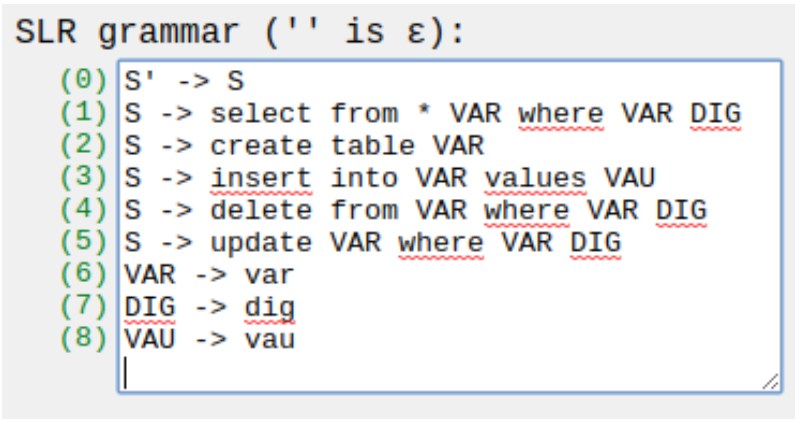

Figura 8 – Gramatica livre de contexto

Para a construção de um analisador sintático é necessário uma tabela de *parsing*, o professor orientou-nos a usar um gerador de tabelas, o escolhido foi

SLR Parser Generator disponível em[2]. A entrada para o SLR Parser Generator é a gramática livre de contexto, e como saída ele nos dá o conjunto *first and follow* que está exibido na figura 9, o conjunto de transições (e o processo para faze-lo) que está exibido na figura 10, e a tabela de parsing.

FIRST / FOLLOW table		
Nonterminal	**FIRST**	**FOLLOW**
S'	{select,create,insert,delete,update}	{$}
S	{select,create,insert,delete,update}	{$}
VAR	{var}	{where,dig,$,values}
DIG	{dig}	{$}
VAU	{vau}	{$}

Figura 9 – Conjunto first e follow

[2] http://jsmachines.sourceforge.net/machines/slr.html

Goto	Kernel	State
	{S' -> .S}	0
goto(0, S)	{S' -> S.}	1
goto(0, select)	{S -> select.from * VAR where VAR DIG}	2
goto(0, create)	{S -> create.table VAR}	3
goto(0, insert)	{S -> insert.into VAR values VAU}	4
goto(0, delete)	{S -> delete.from VAR where VAR DIG}	5
goto(0, update)	{S -> update.VAR where VAR DIG}	6
goto(2, from)	{S -> select from.* VAR where VAR DIG}	7
goto(3, table)	{S -> create table.VAR}	8
goto(4, into)	{S -> insert into.VAR values VAU}	9
goto(5, from)	{S -> delete from.VAR where VAR DIG}	10
goto(6, VAR)	{S -> update VAR.where VAR DIG}	11
goto(6, var)	{VAR -> var.}	12
goto(7, *)	{S -> select from *.VAR where VAR DIG}	13
goto(8, VAR)	{S -> create table VAR.}	14
goto(8, var)	{VAR -> var.}	12
goto(9, VAR)	{S -> insert into VAR.values VAU}	15
goto(9, var)	{VAR -> var.}	12
goto(10, VAR)	{S -> delete from VAR.where VAR DIG}	16
goto(10, var)	{VAR -> var.}	12
goto(11, where)	{S -> update VAR where.VAR DIG}	17
goto(13, VAR)	{S -> select from * VAR.where VAR DIG}	18
goto(13, var)	{VAR -> var.}	12
goto(15, values)	{S -> insert into VAR values.VAU}	19
goto(16, where)	{S -> delete from VAR where.VAR DIG}	20
goto(17, VAR)	{S -> update VAR where VAR.DIG}	21
goto(17, var)	{VAR -> var.}	12
goto(18, where)	{S -> select from * VAR where.VAR DIG}	22
goto(19, VAU)	{S -> insert into VAR values VAU.}	23
goto(19, vau)	{VAU -> vau.}	24
goto(20, VAR)	{S -> delete from VAR where VAR.DIG}	25
goto(20, var)	{VAR -> var.}	12
goto(21, DIG)	{S -> update VAR where VAR DIG.}	26
goto(21, dig)	{DIG -> dig.}	27
goto(22, VAR)	{S -> select from * VAR where VAR.DIG}	28
goto(22, var)	{VAR -> var.}	12
goto(25, DIG)	{S -> delete from VAR where VAR DIG.}	29
goto(25, dig)	{DIG -> dig.}	27
goto(28, DIG)	{S -> select from * VAR where VAR DIG.}	30
goto(28, dig)	{DIG -> dig.}	27

Figura 10 – Conjunto de transições

O algoritmo para reconhecimento do analisador sintático teve como entrada a tabela de símbolos gerada no analisador léxico, mas foi usada apenas a primeira coluna onde estão descritas as regras que reconhecem os *tokens* (fita), também tem como entrada a tabela de *parsing* que teve algumas alterações manuais, como a adição na seguida linha da tabela, a regra onde foi reconhecido o *token*. A tabela de *parsing* está exibida na figura 11.

state	select	from	*	where	create	table	insert	into	values	delete	update	var	dig	vau	$	S'	S	VAR	DIG	VAU
state	131	112	58	127	128	124	130	114	133	129	132	69	76	73	$	S'	S	VAR	DIG	VAU
0	s2				s3		s4			s5	s6						1			
1															acc					
2		s7																		
3						s8														
4								s9												
5		s10																		
6												s12						11		
7			s13																	
8												s12						14		
9												s12						15		
10												s12						16		
11				s17																
12				r6					r6				r6		r6					
13												s12						18		
14															r2					
15									s19											
16				s20																
17												s12						21		
18				s22																
19														s24						23
20												s12						25		
21													s27						26	
22												s12						28		
23															r3					
24															r8					
25													s27						29	
26															r5					
27															r7					
28													s27						30	
29															r4					
30															r1					

Figura 11 – Tabela de Parsing

O primeiro passo no algoritmo de reconhecimento sintático foi percorrer a fita, item por item. Para cada item da fita é procurado sua posição na tabela de *parsing* a procura do estado do *token* e verificar qual função será feita na pilha de execução se será de empilhamento ou redução como vemos código 4 a seguir.

Listing 4 – percorre codigo

```
def quebra(prod, pilha, parse, i, aux):
    if prod.count('s'):
        # EMPILHA
        state = prod.split('s')
        state = int(state[1])
        pilha.append(i)
```

```
        pilha.append(state)
    elif prod.count('r'):
        # REDUZ
        state = prod.split('r')
        state = int(state[1])
        reduz(state, pilha, parse)
        percorreState(parse.iloc[pilha[-1]+1, aux], i, pilha, parse, aux)

    elif prod == 'acc':
        print('linguagem aceita')
        for k in range(2):
            del(pilha[-1])
    elif prod == 0:
        print()
    else:
        print('Erro sintatico por conta de ', prod)
```

O empilhamento vai empilhar o código do *token* e o estado que ele se encontra. A redução presente no código 5 faz o desempilhamento baseado em quantas produções a regra produz, após isso ela faz o salto para o estado dentro da tabela de *parsing* onde se encontra regra.

Listing 5 – percorre codigo

```
def reduz(state, pilha, parse):
    n_prod = np.genfromtxt('sintatic/numeroProd.txt', dtype = 'str', deli
    for i in range(int(n_prod[state,2])*2):
        del(pilha[-1])
    aux = pilha[-1]
    pilha.append(n_prod[state,1])
    pilha.append(salto(parse, aux, n_prod[state, 1]))
```

4 Conclusões

Com esses trabalho foi possível ter um entendimento muito maior das etapas de compilação da análise léxica e semântica, a pesar de termos tido grande dificuldades com a construção da análise sintática mas que a construção em conjunto com as bibliotecas utilizadas foi possível a realização dos passos até então. Como continuidade ainda precisa ser feita a análise semântica, geração de código intermediário e otimização, sendo assim necessária fazer um plano de ação levando em conta a realidade da implementação com as descrições das etapas presentes na descrição do trabalho.

Referências

AHO, A. et al. *Compiladores: principios, técnicas y herramientas*. LTC, 1990. ISBN 9788521610571. Disponível em: <https://books.google.com.br/books?id=gVlMPgAACAAJ>. Citado na página 2.

SETHI, R.; ULLMAN, J. D.; LAM, m. S. *Compiladores: principios, tecnicas e ferramentas*. [S.l.]: Pearson Addison Wesley, 2008. Citado 6 vezes nas páginas 1, 2, 3, 4, 5 e 6.