Flex-PL – Uma linguagem minimalista e os desafios da construção de um compilador em um curso de Ciência da Computação

Felipe de Lima Peressim[1]

[1]Acadêmico Bacharel Ciência da Computação – Instituto Federal de Educação, Ciência e Tecnologia Catarinense (IFC) – Campus Rio do Sul
89160-202 – Rio do Sul – SC – Brasil

fe.peressim@gmail.com

Abstract. *This paper describes the processes, methodologies and the techniques used in the construction of a compiler for a programming language conceived as Flex-Pl and the challenges that have arose over its construction in a compiler discipline as part of a* ***Computer Science*** *course. Being a remarkable and innovative feature about its construction, it is due to the fact that it was made using an interpreted programming language.*

Resumo. *Este artigo descreve os processos, metodologias e as técnicas utilizadas na construção de um compilador para uma linguagem de programação concebida como Flex-pl e os desafios que surgiram no decorrer de sua construção em uma disciplina de compiladores como parte do curso de* ***Ciência da Computação****. Sendo uma característica marcante e inovadora acerca de sua construção, dá-se ao fato que este fora confeccionado usando uma linguagem de programação interpretada.*

1. Introdução

A atividade intelectual de compor um programa de computador é muito parecida com a de compor uma poesia ou uma música. Um programa transmite uma mensagem não apenas para um computador, mas para todos aqueles que pretendem utilizá-lo e modificá-lo (BARTLETT, 2003).

Para realizar a composição de um programa de computador, a fim de expressar uma ideia e transmiti-la ao computador, necessita-se de um meio comum de comunicação de modo que o computador possa realizar uma determinada atividade e/ou resolver um problema ao qual foi designado. Para que isso ocorra efetivamente, é necessário estabelecer uma forma de comunicação entre as partes, através de uma linguagem formal e bem estruturada, da qual os pensamentos humanos possam ser estruturados e modelados de maneira lógica, racional, com certo rigor matemático e sem ambiguidades.

Mesmo com as diversas linguagens de programação existentes as máquinas ainda assim não conseguem interpretar o significado das ideias compostas através delas, o que aparentemente parecem tornar as linguagens de programação desnecessárias. Entretanto, conforme Abelson e Sussman (1996, p.4) "uma linguagem de programação efetiva está muito além de ser apenas um meio de instruir um computador". As linguagens de programação são a ponte de comunicação entre o ser humano e máquina, devido a sua expressividade e suas construções idiomáticas que causam uma proximidade da linguagem natural, entretanto formal, habilitando o escritor e/ou leitor uma maior compreensão acerca das ideias quais pretende-se expressar através do formalismo e meios de abstração que tal linguagem oferece. Tal linguagem, ainda assim deve ser traduzida para uma linguagem que a máquina possa entender, a linguagem de máquina. Para que a comunicação ocorra de fato, é necessário prover de um meio de tradução da linguagem formal para a linguagem de máquina, esse processo de tradução também é conhecido como processo de compilação.

O processo de compilação acontece através de um programa de computador denominado de compilador. Segundo Aho *et al*., (2008, p.1), "um compilador é um programa que recebe como entrada um programa em uma linguagem de programação – a linguagem fonte – e o traduz para um programa equivalente em outra linguagem – a linguagem objeto".

A construção de compiladores é um ramo de sucesso da **Ciência da Computação**. Algumas das razões para isso são a estruturação adequada do problema, o uso criterioso de formalismos e uso de ferramentas sempre que possível (GRUNE *et al.,* 2012). Também é um dos ramos mais antigos, desde que surgiu-se a necessidade de comunicar-se com as máquinas de maneira efetiva e sem as dificuldades da compreensão da linguagem de máquina devido ao idioma binário que esta utiliza. Essas e outras razões, deram motivação para a escrita deste artigo o qual objetiva descrever a metodologia, os processos e as técnicas utilizadas no desenvolvimento de um compilador para uma linguagem de programação concebida como *Flex-Pl*[1] e os desafios que surgiram no decorrer de sua construção que ocorreram durante a participação do autor na disciplina de compiladores em um curso de **Ciência da Computação**. Sendo uma característica marcante e inovadora acerca de sua construção, dá-se ao fato que este fora confeccionado usando uma linguagem de programação interpretada.

2. Justificativa

O estudo de compiladores é uma disciplina fundamental para a **Ciência da**

1 *Flex-Pl* é um acrônimo para *Flexible Programming Language (*Linguagem de programação Flexível*).* Este nome foi concebido pelo autor deste artigo, devido ao projeto inicial da linguagem que previa um compilador para uma linguagem minimalista de sintaxe simples com construções idiomáticas próximas de uma linguagem natural utilizando palavras em inglês na maior parte de suas construções no lugar de símbolos abstratos.

Computação devido a sua maturidade, uma vez que as teorias acerca desta, conforme Grune *et al.* (2012, p. 7), vem se desenvolvendo desde 1960 e também por fazer o uso de rigorosos conceitos e formalismos desenvolvidos desde então. Sendo esta uma extensão da disciplina de linguagens formais e autômatos, presente na maioria dos cursos de **Ciência da Computação**, os conceitos teóricos estudados nas linguagens formais encontram a prática na disciplina de compiladores, sendo que os aspectos práticos podem ser entendidos sem aprofundar-se muito na teoria.

Um compilador é um programa bastante complexo, que pode ter de 10.000 a 1.000.000 linhas de código. Desenvolver um programa deste tipo, não é uma simples tarefa, e um indivíduo único jamais escreverá um compilador completo (LOUDEN, 2004). A construção de um compilador envolve o uso e desenvolvimento de diversos algoritmos e estruturas de dados como tabelas *hash,* pilhas, árvores, tabela de símbolos, programação dinâmica e muitos outros, os quais embora possam ser estudados separadamente acabam por serem mais educacionalmente valiosos se forem estudados em um contexto mais significativo (GRUNE *et al.,* 2012).

O objetivo geral de desmistificar o processo da construção de um compilador para linguagem *Flex-Pl,* dá-se justamente a complexidade do processo metodológico e de desenvolvimento de um compilador no geral, com o intuito de auxiliar os estudantes que estão se deparando pela primeira vez com a construção de compiladores, de modo que as experiências e os desafios encontrados ao longo do desenvolvimento deste possam direcionar e contribuir para o sucesso de outros projetos de compiladores.

No decorrer construção deste, durante a disciplina de compiladores, algoritmos que dizem respeito a análise léxica e sintática, foram desenvolvidos separadamente sem o uso de geradores, para reconhecerem apenas pequenas partes da linguagem, servindo como exercício prático para os alunos devido sua importância e ampla aplicabilidade que esses possuem em diversas áreas da computação, principalmente aquelas que dizem respeito ao reconhecimento de padrões em cadeias de caracteres. Também objetivando a compreensão dos alunos acerca das fases e processos de desenvolvimento de um compilador. Além disso, para o desenvolvimento final do compilador, foram apresentados e utilizados programas geradores, de analisadores léxicos e sintáticos, que foram integrados com o analisador semântico e gerador de código *Assembly*.

3. Fundamentação Teórica

O processo de tradução de um programa fonte para um programa objeto é um processo que passa por várias fases as quais são destrinchadas em duas partes, análise e síntese. A Figura 1 exibe uma ilustração sobre as fases deste processo.

Na parte de análise, o programa fonte é subdividido em partes constituintes e sobre estas é imposta uma estrutura gramatical. Posteriormente essa estrutura é usada para criar uma representação intermediária do programa fonte. Nesse estágio, erros relativos

a sintaxe e semântica do programa podem ser detectados e caso isso ocorra, então a análise precisa prover de mensagens que auxiliem o usuário a tomar medidas corretivas acerca dos erros encontrados. Além disso, são coletadas informações sobre o programa fonte, tais como variáveis e funções declaradas bem como seus tipos de dados e outras informações que sejam pertinentes ao compilador conforme arquitetado em sua construção as quais são armazenadas em uma estrutura de dados chamada tabela de símbolos, que é utilizada a posteriori junto de sua representação intermediária para a parte de síntese (AHO *et al.,* 2008).

O programa objeto desejado é construído no estágio de síntese a partir da representação intermediária e das informações na tabela de símbolos. A parte da análise normalmente é chamada de *front-end* do compilador; a parte de síntese é o *back-end* (AHO *et al.,* 2008). Nessa parte, o compilador usa as informações obtidas através da fase anterior para gerar o código objeto, ou seja, o código final o qual pode ser interpretado por um computador.

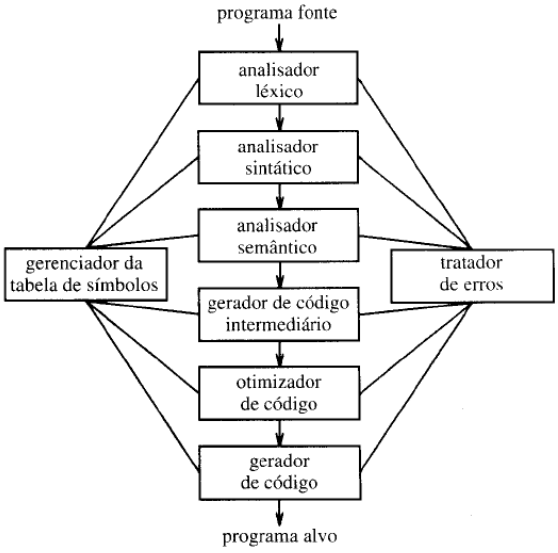

Figura 1: Fases de um compilador. Fonte: (AHO *et al.,* 2008)

3.1. Análise Léxica

A análise léxica é a primeira fase de um compilador, também chamada de *scanning*. Nessa fase é realizada a leitura do código fonte, caractere por caractere, os quais são traduzidos para uma sequência de símbolos léxicos, unidades significativas do programa também chamadas de *tokens.* (PRICE; TOSCANI, 2008)

As palavras identificadas através do analisador léxico, são sequências significativas que fazem parte da linguagem fonte, que geralmente dizem respeito às suas construções idiomáticas as quais englobam as sequências que são válidas em seu

contexto, como identificadores e palavras chaves. Essas palavras são identificadas pelo analisador léxico através de uma representação abstrata de máquina conhecida como autômatos, os quais são representados algebricamente através de expressões regulares devido ao seu poder de reconhecer padrões em textos. Podendo estas descrever um conjunto infinito de cadeias de caracteres e realizar o casamento das mesmas. Por exemplo, a expressão regular denotada por **[a-z]*** gera um conjunto infinito de palavras que contém qualquer combinação de todas as letras minúsculas do alfabeto latino. Conforme Hopcroft, John E; Ullman, Jeffrey D; Motwani, Rajeev, (2002, p.97), ambos os mecanismos representam exatamente o mesmo conjunto de linguagens, as linguagens regulares, sendo uma apenas representação alternativa do outro.

Os Autômatos Finitos também conhecidos como Máquinas de Estados Finita é um formalismo matemático que descreve tipos particulares de algoritmos, o qual é representado através de um ou mais grafos onde seus estados são representados por nodos e os arcos representam as transições entre estados. Os algoritmos gerados a partir dos autômatos finitos, são utilizados para reconhecer padrões em sequências de caracteres, e portanto podem ser utilizados para construir sistemas de análise léxica A Figura 2 ilustra um exemplo de autômato que reconhece as palavras-chave *if* e *int*, onde o triângulo indica o estado inicial, e os estados finais são distinguidos por uma circunferência concêntrica (LOUDEN, 2004), (PRICE; TOSCANI, 2008).

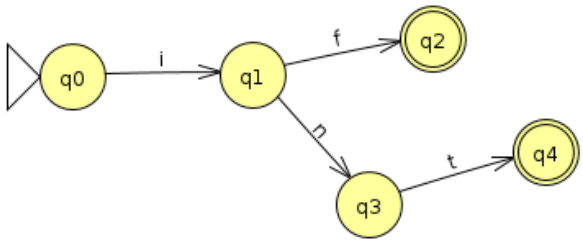

Figura 2: Autômato que reconhece as palavras-chave *if* e *int.* Fonte: Acervo do autor.

3.2. Análise Sintática

A análise sintática também conhecida como *parsing*, compõe a segunda fase de um compilador. Sua designação dá-se a verificação das construções utilizadas no programa-fonte estão gramaticalmente corretas (PRICE; TOSCANI, 2008). Essa verificação é feita através de regras gramaticais atribuídas a linguagem alvo. No contexto de linguagens formais, uma gramática é designada como um mecanismo que gera as sentenças (ou palavras) de uma linguagem. No desenvolvimento de compiladores, o mais comum é a utilização de uma classe específica de gramáticas, as gramáticas livres de contexto.

Uma gramática livre de contexto é a denotação de um formalismo para expressar

definições recursivas de linguagens. Uma gramática consiste em uma ou mais variáveis que representam classes de cadeias de caracteres, isto é, linguagens. (HOPCROFT ; ULLMAN; MOTWANI, 2002)

Formalmente, segundo Price e Toscani (2008, p.18) uma gramática livre do contexto é definida pela quádrupla (N, T, P, S)

onde

- N é um conjunto de símbolos não terminais (variáveis)
- T é um conjunto de símbolos terminais (constantes) e $T \cap N = \emptyset$, que serão os *tokens* gerados pela análise léxica;
- P é um conjunto de regras de produção (regras sintáticas)
- S é o símbolo inicial da gramática $(S \in N)$

O mecanismo de geração de sentenças é dado pelas regras de produção, onde geralmente uma regra de produção é da forma $A \rightarrow \alpha \beta B \gamma$ onde A e B são não-terminais (variáveis) e α , β e γ são os terminais (constantes). Esse tipo de sentença pode ser lida como: A deriva $\alpha \beta B \gamma$. O lado esquerdo de uma regra gramatical pode aparecer mais de uma vez, o que indica que o não terminal têm mais de uma regra que são alternativas deste, sendo que essas alternativas podem ser separadas pelo caractere "|", o qual tem seu significado análogo a disjunção "ou". Um exemplo de uma regra gramatical dada por uma produção em uma linguagem de programação é dada abaixo:

$$E \rightarrow E+T \mid E-T \mid T \quad (1)$$

$$T \rightarrow T*F \mid T/F \mid F*F \mid F \quad (2)$$

$$F \rightarrow (E) \mid id \quad (3)$$

Tais regras de produção dizem respeito ao reconhecimento sintático de expressões aritméticas respeitando a precedência de operadores, conforme o padrão da matemática. Observa-se também que, para cada não-terminal, existe mais de uma regra diferente, separadas por "|". Através dessa descrição, é possível construir uma árvore de análise sintática, a qual é útil para a derivação das regras gramaticais no processo de compilação. Por exemplo, uma sentença do tipo $id*id$ gera a árvore mostrada na Figura 3:

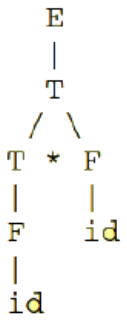

Figura 3: Árvore de análise sintática para a expressão id * id. Fonte: Acervo do autor.

3.3. Análise Semântica e demais fases

A análise semântica requer verificações que estão além do poder das gramáticas livres de contexto e demais algoritmos de análise sintática (LOUDEN, 2004). O tipo de informação qual deseja-se verificar através da análise semântica está ligada integramente ao significado que têm as construções gramaticais de uma linguagem em particular. Em algumas linguagens por exemplo, é necessário que ocorra a verificação de tipos de dados durante a fase de compilação. Também verifica-se se as variáveis foram declaradas e inicializadas antes de serem usadas, e se os dados usados em uma expressão são objetos computacionais relativos ao mesmo tipo de dado, ou um tipo de dado que seja compatível entre ambos. Tais verificações são feitas na maioria das linguagens de programação, entretanto não são regras que devem ser seguidas, obviamente, a semântica tem significados diferentes em cada tipo de linguagem, levando a regras diferentes de verificação, ou seja, a semântica das linguagens podem variar, como por exemplo as linguagens de programação *Python* e *PHP* as quais são fracamente tipificadas devido a seu sistema de tipagem dinâmica, em contraste, *Java* e *C* são linguagens fortemente tipificadas devido a sua tipificação estática. Além destas, diversas outras diferenças existem e não sendo particulares a uma linguagem em específico, porém essas diferenças não fazem parte do escopo deste artigo e não são discutidas aqui.

As três fases de análise léxica, sintática e semântica compõe a primeira parte (análise) do processo de compilação. Durante estas fases, foram geradas a tabela de símbolos e a árvore abstrata de análise sintática as quais são utilizadas nas fases subsequentes do processo de compilação.

Embora a maioria dos compiladores modernos, passem por todas as fases descritas na Figura 1, algumas dessas fases podem ser omitidas, como a otimização de código e a geração de código intermediário. Na fase de otimização de código ajustes são feitos em relação a redundâncias que possam existir no código, redução no número de variáveis e o pré-cálculo de algumas expressões para evitar o uso excessivo de registradores e memória. Além dessas, outras otimizações são especificadas de acordo

com o projeto da linguagem.

Geralmente nessa fase, é gerado código objeto, no contexto do compilador para *Flex-Pl* gerou-se código em inguagem de montagem (*Assembly*) diretamente sem realizar otimizações como a geração do código de três endereços por exemplo. Quando ocorre a geração de código objeto em *Assembly,* posteriormente este pode ser ligado a bibliotecas providas pelo sistema operacional e por fim gerar-se o código de máquina sobre o código intermediário gerado no processo de compilação se especificado no compilador em si, o qual pode-se então, ser executado na máquina alvo.

4. Descrição dos processos de construção de um compilador para a linguagem Flex-pl

Antes do processo da construção do compilador para a linguagem *Flex-Pl* iniciar-se, no sentido prático, fora elaborado uma descrição formal, porém breve acerca da linguagem em si. Nesta fora descrito as construções idiomáticas e sintáticas da linguagem, tais como as principais palavras chaves, tipos de operadores aritméticos e lógicos, operadores de atribuição, controles de fluxo e desvios condicionais e a exibição de exemplos descritivos relativos a sintaxe que vão desde o formato de declaração de variáveis até exemplos de funções recursivas, como o cálculo do fatorial de dado número *n* e a sequência de Fibonacci com sua recorrência abstraída na sintaxe da *Flex-Pl.*

Inspirada pelas linguagens de programação *Scheme, Python* e *C,* em sua primeira especificação formal, a construção da linguagem *Flex-Pl* objetivava a implementação de uma linguagem que fosse minimalista, de sintaxe simples e descritiva a fim de que suas construções idiomáticas pudessem priorizar a legibilidade e facilidade na produção e análise de código fonte. A sintaxe descrita tinha similaridades com a linguagem de programação *Python*, minimalismo e expressividade de *Scheme* e tipagem estática inspirada pela linguagem *C*. Entretanto, no decorrer de seu desenvolvimento mudanças relativas na sintaxe da *Flex-Pl* ocorreram, devido a determinadas construções idiomáticas presentes nesta que necessitavam ser tratadas de forma programática pelo compilador devido as **gramáticas livres de contexto** não terem poder suficiente para tratá-las, como a indentação orientada a tabulações/espaços e comandos não terminados por ponto e vírgula, por exemplo no contexto da gramática da *Flex-Pl* uma função é referida através um identificador seguido de parênteses, com ou sem argumentos, e uma expressão poder ser terminada por um identificador tornando a gramática ambígua por essa não saber se uma função $f(x)$ é uma ou duas expressões. Com o propósito de construir um compilador clássico, baseado nos formalismos presentes nas linguagens formais, como por exemplo, a descrição da linguagem totalmente através das gramáticas livres do contexto, ocorreram mudanças para atingir os objetivos da disciplina com

sucesso.

A seguir é apresentado um trecho de código da linguagem *Flex-Pl,* o qual exibe as principais palavras chaves e algumas construções através de um algoritmo que computa a série de *Fibonacci* através da definição da função *fib*.

Listagem 1: Exemplo de gerador da série de Fibonacci abstraído em *Flex-pl na* sintaxe final da linguagem. Fonte: Acervo do autor.

```
0 define int fib(int n) {
1     if (n equal 0) {
2         return 0;
3     }
4     else if (n equal 1) {
5         return 1;
6     }
7     else {
8         return fib(n - 1) + fib(n – 2);
9     }
10 }
```

Conforme o código da Listagem 1 (um), observa-se que a linguagem provê meios de abstração para combinar simples ideias em ideias mais compostas através da definição e chamada de funções. Tipos de dados devem estar presentes na declaração de variáveis, nos parâmetros formais e na declaração da função, sendo que para declarar uma função deve-se usar a palavra reservada *define.* Declarações de variáveis, atribuição e expressões devem terminar com ponto e vírgula “;” e a função deve sempre retornar um valor, podendo esta realizar chamadas recursivas. O código em si é auto explicativo, devido a expressividade da linguagem e sua similaridade com a linguagem *C,* esta recebe um parâmetro o qual é um inteiro $n \geq 0$ e calcula-se o fatorial deste número através de chamadas recursivas, tendo-se dois casos base de parada.

4.1. Processos de construção do compilador

Todas as principais fases do processo de construção foram atendidas exceto as partes que dizem respeito a otimização e geração de código de três endereços devido ao foco estar na construção de um pequeno compilador funcional e não em um projeto grande que pudesse ser falho devido a grande complexidade que este pudesse ter, isso já considerando a complexidade de um compilador simples como este.

Para a realização dessa construção, bem com a implementação dos algoritmos, e usou-se a linguagem de programação ***Python 3.6.5***. Na geração de analisadores sintático e léxico usou-se a implementação ***Ply*** *(*DABEAZ, 2018) e a linguagem de montagem ***Assembly*** para plataforma ***x86*** para a geração de código alvo.

4.2. Ply

Durante a fase de implementação dos algoritmos, imediatamente usou-se um gerador de analisador léxico e sintático de modo a acompanhar o projeto em cada fase, para que não fosse necessário passar por todas as fases novamente.

O gerador usado é uma implementação das ferramentas mais usadas na construção de compiladores *lex* e *yaac* (LEVINE; MASON; BROWN, 1992) escrita em *Python*. Concebido como *Ply* e desenvolvido como uma ferramenta de suporte para um curso de compiladores em 2001 na universidade de Chicago por David M. Beazley, seu principal objetivo é manter suas funcionalidades fiéis a essas ferramentas o qual este fora baseado. Este incluindo suporte para geração de tabelas de análise sintática LALR(1), validação de dados de entrada extensiva, relatório de erros e diagnósticos *(*DABEAZ, 2018).

O gerador *Ply* consiste de dois módulos distintos; *lex.py* e *yaac.py,* ambos são encontrados em um pacote *Python* denominado *ply*. O *lex.py* é usado para quebrar o texto em uma coleção de *tokens* especificados por uma coleção de expressões regulares. O *yaac.py* é usado para reconhecer a sintaxe da linguagem que tenha sido especificada na forma de gramática livre do contexto.

Ambos são ferramentas que trabalham juntas. Especificamente, *lex.py* prove uma interface externa através da função *token()*, a qual retorna o próximo *token* válido do fluxo de entrada. Essa função é chamada pelo *yaac.py* de maneira repetida para recuperar os *tokens* e invocar as regras gramaticais. A saída do *yaac.py* geralmente é uma **Árvore Sintática Abstrata**. Entretanto usá-la fica a critério do usuário se este desejar pode usar o *yacc.py* para implementar um compilador simples de uma passada. Um compilador é dito de uma passada quando este não gera código intermediário como no caso do código de três endereços por exemplo, ou seja, o código objeto é gerado diretamente sem que ocorram otimizações através da geração de código intermediário (AHO *et al.,* 2008).

4.3. Análise Léxica

Na primeira fase do compilador para *Flex-Pl* desenvolveu-se um analisador léxico usando-se o autômato finito determinístico ilustrado na Figura 4 como base. Para isso, utilizou-se a ferramenta *Jflap* (JFLAP, 2018), como auxiliadora onde fora desenhado o autômato, de modo que este pudesse auxiliar visualmente na codificação. A Figura 4 exibe tal autômato.

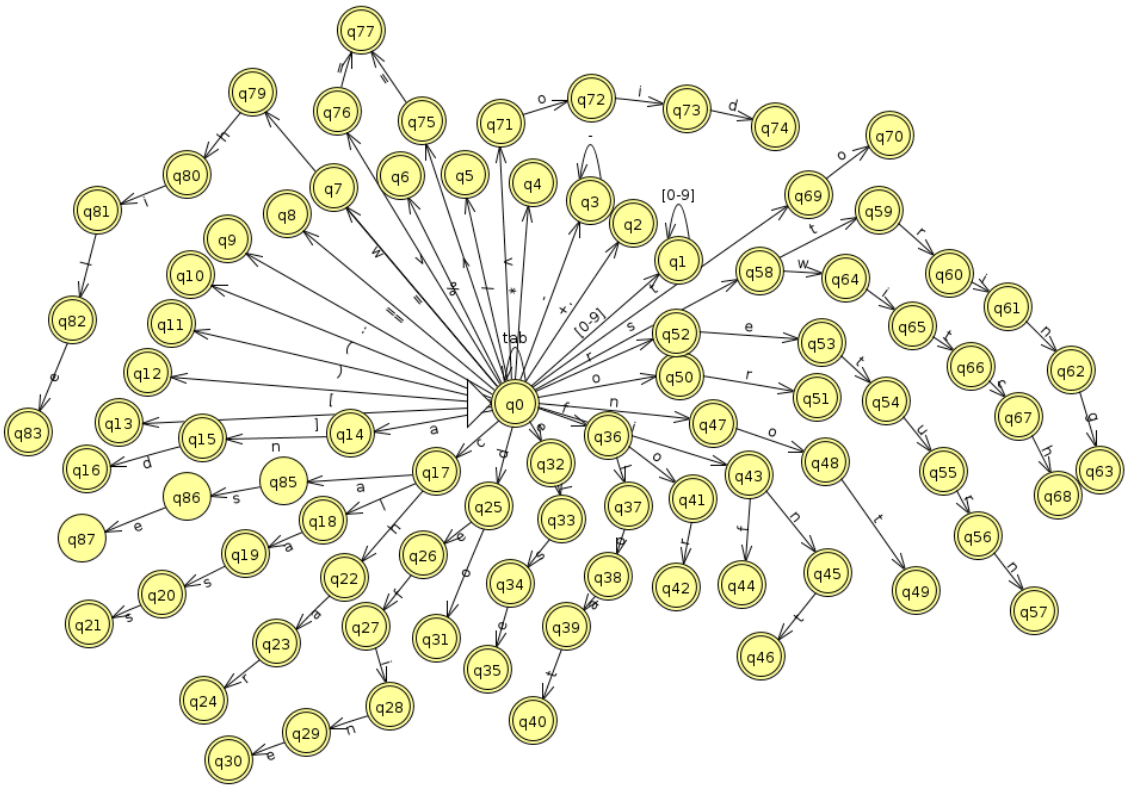

Figura 4: Representação em forma de grafo do Autômato Finito que reconhece palavras pertencentes a linguagem *Flex-PI.*

O autômato da Figura 4 foi implementado utilizando-se a linguagem de programação *Python* com objetivos de aprendizado acerca do funcionamento de um reconhecedor léxico como citado anteriormente. Para isso fez-se o uso extensivo da estrutura de dados *dicionário,* também mais comumente conhecida como *Hash Table*, presente na linguagem *Python*. Esta utilizada para representar a função de transição do autômato onde o estado atual concatenado com os caracteres obtidos através da varredura de forma individual, representa uma chave da qual o valor obtido através dela é uma transição para o próximo estado, enquanto os caracteres lidos forem válidos.

A Listagem 2 a seguir ilustra a geração do autômato através do uso de uma função auxiliadora a qual gera as transições entre os estados do autômato de acordo com os padrões definidos, sendo neste caso para o reconhecimento de operadores aritméticos, de comparação o operador lógico *and* e alguns casos especiais de identificadores.

Listagem 2: Trecho de código que realiza a geração dos estados do autômatoe transições para reconhecimento de palavras da linguagem Flex-PI. Fonte: Acervo do Autor.

```
0 #Operadores aritméticos, igualdade, atribuição parenteses, colchetes e dois
pontos.
1 generator.addStateTransitionState('q0', ['q2', 'q3', 'q4', 'q5', 'q6', 'q7',
'q9', 'q10', 'q11', 'q12', 'q13'], listOfTransitions=['+', '-', '*', '/', '%',
'=', ':', '(', ')', '[', ']'])
2 # ==
3 generator.addStateTransitionState(['q7'], ['q8'], listOfTransitions=['='])
4 # Operador lógico AND
5 generator.addStateTransitionState(['q0', 'q14', 'q15'], ['q14', 'q15',
'q16'], listOfTransitions=['a', 'n', 'd'])
6 # ID
7 generator.addStateTransitionState('q14', 'q84', addLowerCase=True,
addUpperCase=True, addZerothroughNine=True, listOfConstraints=['n'])
7 generator.addStateTransitionState('q15', 'q84', addLowerCase=True,
addUpperCase=True, addZerothroughNine=True, listOfConstraints=['d'])
9 generator.addStateTransitionState('q16', 'q84', addLowerCase=True,
addUpperCase=True, addZerothroughNine=True)
```

Tal trecho de código adiciona determinados padrões na *Hash Table* a serem reconhecidos pelo autômato, denominados de lexemas. Aos lexemas serem reconhecidos pelo programa, são alcançados os estados finaisque são utilizados como entradas para outra *Hash Table* a qual tem como valores seus respectivos *tokens*, que são programados manualmente no sentido da não utilização de geradores para tal, conforme mostra na Listagem 3, e usados através do algoritmo mostrado na Listagem 4 o qual simula o movimento do autômato.

Listagem 3: Definição dos *Tokens* na *Hash Table.* Fonte: Acervo do autor.

```
0 tokens = {
1       'q1': 'CONSTANT',
2       'q2': 'ARITHMETIC_OP',
3       'q3': 'ARITHMETIC_OP',
4       'q4': 'ARITHMETIC_OP',
5       'q5': 'ARITHMETIC_OP',
6       'q6': 'ARITHMETIC_OP',
7       'q7': 'ASSIGNMENT',
8       'q8': 'EQUAL',
9       'q9': 'TWO_POINTS',
10       'q10': 'OPEN_PAREN',
11       'q11': 'CLOSE_PAREN',
12       'q12': 'OPEN_BRACKETS',
```

```
13       'q13': 'CLOSE_BRACKETS',
14       'q14': 'ID',
15       'q15': 'ID',
16       'q16': 'LOGICAL_OP', # and
…
```

Listagem 4: Algoritmo genérico que simula o movimento do autômato. Fonte: Acervo do autor.

```
0 def move(s, c):
1     global state
2     try:
3         return state[s+c]
4     except KeyError:
5         print (s)
6         return None
7
8 def lexicalAnalyzer(c):
9     lexema = ""
10     tokenLexema = ''
11     tokens = []
12     c = parseInputAndPrepareInput(c)
13
14     for string in c.split():
15         word = string[:]
16         s = 'q0'
17         lexema = ""
18         while (len(word) > 0 and s is not None):
19             lexema += word[0]
20             s = move(s, word[0])
21             word = word[1:]
22         if s in tokensTable:
23            tokens.append(tokensTable[s])
24             tokenLexema += "<" + tokensTable[s] + ","+""+ lexema +
">\n"
25         else:
26              return 'Lexical error. Unknown symbol.'
27    return tokens
```

O algoritmo mostrado na Listagem 4, simula qualquer autômato finito determinístico com base nos estados pré-programados em uma *Hash Table*, neste caso, por tratar-se da linguagem *Python*, a estrutura de dados dicionário foi o suficiente para tal. A função *lexicalAnalyser* recebe como entrada o código fonte, prepara a entrada separando caractere por caractere para realizar o reconhecimento de padrões através do autômato simulado pela função *move.* Ao final o algoritmo verifica se o estado atual é um estado final, caso este seja o programa retorna uma tabela com os *tokens* e *lexemas*, caso contrário sinaliza um erro.

O autômato em si, é simulado através de uma função denominada de *move* a qual recebe como entrada o estado atual e o próximo caractere da entrada. A função tenta retornar um valor presente no dicionário através da concatenação do estado atual com o caractere corrente, e, verificar se esta combinação é uma chave válida. Então a transição do autômato é realizada de tal modo que o próximo estado é retornado para a função chamadora, caso contrário um valor nulo é retornado. Portanto a chamadora saberá se deverá sinalizar ou não um erro com base nesse valor de retorno.

Mesmo sendo uma linguagem com poucos padrões, ainda assim escrever um analisador léxico manualmente acaba sendo uma tarefa árdua e dificultosa correção de erros devido a grande quantidade de estados presentes no autômato, sendo 83 estados ao todo no caso do autômato da Figura 4. Em uma linguagem de programação completa torna-se inviável a construção de um autômato de forma manual, nesse caso torna-se ideal o uso de um gerador, que através de definições regulares possa gerar um autômato equivalente. Por fins didáticos fez-se o uso tanto da implementação manual quanto a de um gerador.

No *lex.py* os padrões da *Flex-Pl* que são convertidos para *tokens* são identificados através de uma série de expressões regulares definidas utilizando-se o módulo de expressões regulares presentes na *Python.* A Listagem 5 exibe a declaração de alguns padrões. Palavras reservadas têm os *tokens* definidos a parte através de um dicionário. Os *tokens* sem o prefixo "t" são definidos em uma lista a qual é concatenada com os valores presentes no dicionário das palavras chaves para uso posterior do analisador léxico.

Listagem 5: Declarações de definições regulares da *Flex-Pl*. Fonte: Acervo do autor.

```
108 # Assignment
109 t_EQUALS  = r'='
111 # Integer literal
112 t_INTCONST = r'\d+'
114 # Floating literal
115 t_FLOATCONST = r'\d+[.]\d+'
```

```
116 # String literal
117 t_STRINGCONST = r'\"([^\\\n]|(\\.))*?\"'
118 # Character constant 'c' or L'c'
119 t_CHARCONST = r'\'([^\\\n]|(\\.))*?\''
120
121 # Rule for identificator
122 def t_ID(t):
123     r'[a-zA-Z_][a-zA-Z0-9_]*'
124     t.type = reserved.get(t.value,'ID')# Check for reserved words
125     return t
```

Conforme observa-se na Listagem 5, as definições de *tokens* tem um prefixo em seus nomes, a letra "t", a qual é usada pelo *Ply* como uma forma padrão de invocação destes para uso interno do reconhecedor léxico. A definição de identificadores é um caso especial devido as palavras reservadas presentes na linguagem serem determinadas por um padrão idêntico. Para identificadores é definida uma função a qual é invocada quando tem-se um *token* do tipo identificador para que o analisador possa determinar se o padrão reconhecido é de fato um identificador ou uma palavra-chave. Outras funções também foram definidas com o propósito de identificar outros padrões e detectar possíveis erros léxicos provendo de informações como números de linha e coluna onde os erros ocorreram.

Por fim para a realização de testes é possível invocar o método *lex.lex(data),* alimentando-o com um *buffer* contendo código fonte, o qual retorna uma instância do *lexer* que pode ser usada em um laço, qual a cada iteração retorna um par contendo o *token* identificado e seu respectivo lexema. Uma vez que tenha realizado os testes, não se torna necessário nenhum tipo de invocação, o analisador sintático *yaac.py* sabe como invocar o *lex.py* para fazer a requisição dos *tokens* uma vez que estes são necessários.

4.4. Análise Sintática

Antes de realizar as definições gramaticais através do *Ply*, construiu-se um algoritmo genérico para realizar a análise sintática através do uso parcial do algoritmo SLR (AHO *et al.,* 2008) o qual realiza a análise sintática através de uma tabela criada usando para isso a tabela gerada pelo software *Gals* (GESSER, 2003). O *Gals* utiliza o mesmo algoritmo para gerá-la através de definições regulares e a gramática em sí. *I*mplementando-a em *Python* manualmente e usando o algoritmo implementado para realizar a análise sintática com base nessa tabela através do uso da estrutura de dados *pilha*. A Listagem 6 exibe tal algoritmo.

Listagem 6: Algoritmo genérico para *SLR Parsing.* Fonte: Acervo do autor

```
194 def slrParser(w):
195     w.append('$')
196     stack = ["0"]
197     a = w[0]
198     b = a
199     while True:
200         stateTopOfStack = stack[-1]
201         try:
202             action = actionParseTable[stateTopOfStack + a]
203         except KeyError:
204             action = "ERROR"
205         if action[0]  == "SHIFT":
206             print("SHIFT  " +  a)
207             stack.append(action[1])
208             w = w[1:]
209             a = w[0]
210         elif action[0] == "REDUCE":
211             productionNumber = action[1]
212             b = productions[int(productionNumber)].split()[0]
213             popStackSymbols(stack, productionNumber)
214             topOfStack = stack[-1]
215             stack.append(goto[topOfStack +productionNumber])
216             print("Reduce  to  "  +  b  +  "  according  to  "  +
productions[int(productionNumber)])
217         elif action[0] == "ACCEPT":
218                   print("Expression accepted")
219                   break
220         else:
221                   print("Sintax AnalysesParsing Error")
222                   break
223
224 def popStackSymbols(stack, productionNumber):
225     for i in range(itemsToPop[productionNumber]):
226         stack.pop()
227     return
```

O autômato codificado manualmente expressado na Listagem 4, alimenta a função *slrParser* com uma lista a qual contém os *tokens* separados e na ordem em que estes surgiram no decorrer da análise léxica. O símbolo especial *$* é adicionado ao final da lista para que este possa indicar que a análise sintática ocorreu com sucesso, conforme indicado nas bibliografias as quais este artigo é fundamentado. O topo da pilha é inicializado com o valor 0 (zero) que indica o estado inicial do *parser*. A tabela de ação mostrada na Figura 5 foi codificada através do uso da estrutura de dados dicionário, formando a chave do dicionário os estados da tabela concatenados com os possíveis *tokens* tendo como valores tuplas as quais têm como elementos a ação que a tabela deve realizar e o estado que ser inserido na pilha. Quando a ação é do tipo *REDUCE*, o valor da tupla é uma chave para outro dicionário que retorna um inteiro sendo este uma chave para outro dicionário que tem como valor inteiro, a quantidade de itens que devem ser removidos da *pilha*. Esses itens são removidos através da função *popStackSymbols* definida na linha 224 da Listagem 6.

ESTADO	AÇÃO							
	$	id	tab	logicalop	digit	newline	"("	")"
0	REDUCE(1)	SHIFT(17)	SHIFT(7)	-	SHIFT(21)	REDUCE(26)	SHIFT(15)	-
1	ACCEPT	-	-	-	-	-	-	-
2	-	-	-	-	-	SHIFT(22)	-	-
3	-	SHIFT(17)	SHIFT(7)	-	SHIFT(21)	REDUCE(26)	SHIFT(15)	-
4	-	SHIFT(17)	SHIFT(7)	-	SHIFT(21)	REDUCE(26)	SHIFT(15)	-
5	-	SHIFT(17)	SHIFT(7)	-	SHIFT(21)	REDUCE(26)	SHIFT(15)	-
6	-	-	-	-	-	REDUCE(22)	-	-
7	-	SHIFT(17)	SHIFT(7)	-	SHIFT(21)	REDUCE(26)	SHIFT(15)	-

Figura 5: Tabela de análise sintática parcial gerada pelo software Gals. Fonte: Acervo do autor.

A função *GOTO* neste contexto não faz o papel de uma função em si, porém também é utilizada como um dicionário, sempre que ocorre uma ação do tipo *REDUCE*, após os itens serem removidos da *pilha*, o estado no topo da *pilha* é concatenado com o número da produção (as produções foram numeradas para serem utilizadas nesse algoritmo, cada produção existente bem como suas alternativas foram numeradas para tal propósito) tornando-se uma chave para um valor o qual indica o próximo estado que deve ser inserido na *pilha*.

Como já citado anteriormente, o propósito dessa implementação serviu para que os alunos pudessem compreender o funcionamento do algoritmo considerado o mais simples dos métodos LR (AHO *et al.*, 2008), o qual ocorreu com sucesso, e por seguinte continuou-se a implementação do compilador para *Flex-Pl* através do auxílio do *Ply*.

As regras sintáticas são definidas no *Ply* através da definição de uma gramática livre de contexto usando-se a notação/Formalismo de **Backus-Naur** – ***BNF*** – (SEBESTA, 2011), com a distinção que a separação do não terminal da produção é feito através de dois pontos invés de uma dupla de dois pontos seguidos pelo sinal de igualdade. A separação das alternativas para uma produção mantém-se a mesma.

Essas regras/produções são definidas dentro de funções que usam um prefixo "p", como no caso do analisador léxico, o *yaac.py* usa esse prefixo internamente para realizar a derivação das produções e realizar outras computações internamente relativas a análise sintática. Além das regras de produção, também é possível realizar ações semânticas e gerar a árvore sintática abstrata além realizar outras computações que auxiliem e atendam a especificidades da linguagem que necessitem de computações a parte que a gramática livre de contexto não possa suprir. O analisador *Ply* ainda possibilita definir a precedência operadores, de modo que o usuário possa escrever a gramática sem se preocupar em definir uma gramática que seja derivada com respeito a essas regras.

Listagem 7: Definições parciais da gramática da *Flex-Pl*. Fonte: Acervo do autor.

```
288 def p_term_mod(p):
289     'term : term MOD factor'
290     p[0] = BinaryOp(p[1], p[2], p[3])
291     pass
292 
293 def p_term_factor(p):
294     'term : factor'
295     p[0] = p[1]
296     pass
297 
298 def p_factor_num(p):
299     '''factor : number
300               | ID
301               | factor_expr
302               | function_call'''
303     p[0] = p[1]
304     pass
305 
306 def p_factor_expr(p):
307     'factor_expr : LPAREN expression RPAREN'
```

```
308     p[0] = p[2]
309     pass
```

O analisador sintático oferece uma forma de construir a árvore sintática abstrata de uma maneira muito fácil, a qual ocorre durante a derivação das regras gramaticais. Cada função que rege uma regra recebe como parâmetro uma lista onde cada posição referencia uma parte da regra gramatical, por exemplo, a posição 0 (zero) refere-se ao lado esquerdo da produção atual, a posição 1 (um) refere-se ao primeiro termo do lado esquerdo e assim por diante. Nessa lista é possível fazer atribuições em suas respectivas posições de objetos relativos a cada regra gramatical definidos pelo usuário, os quais podem tanto ser um único objeto genérico como objetos específicos de acordo com o que o usuário escolheu.

Na figura 10, na função *p_term_mod* um objeto do tipo *BinaryOp* é instanciado e salvo na posição 0 (zero) da lista, enquanto os outros termos do lado direto são provenientes de outras regras que já foram derivadas, ou seja, nas outras posições já têm objetos que são inseridos como nodos na instância do objeto *BinaryOp.* Sendo que essas inserções são feitas internamente pelo próprio *Ply*.

Para usar o analisador sintático, deve-se instanciar o *parser* através de uma chamada ao método *yaac* e por conseguinte alimenta-se a instância com o código fonte, faz-se uma chamada ao método parser o qual internamente alimenta o analisador léxico e faz o pedido de *tokens* a este enquanto vai realizando a derivação. Se a derivação for realizada com sucesso uma instância para a árvore sintática abstrata é retornada como valor, caso contrário um valor nulo é retornado.

Mensagens de erro com informações acerca onde o erro ocorreu também são definidas através de funções com o prefixo “p”, uma função especial chamada denominada de *p_erro* pode-se ser definida pelo usuário como uma função genérica para erros que ocorram durante a derivação das regras. Se o usuário quiser erros mais detalhados, este deve definir novamente as mesmas regras gramaticais com um terminal denominado de *error* do lado mais direito da produção e o nome da função de ativação com o pós-fixo *error.*

Uma vez que o analisador sintático é executado pela primeira vez, algumas mensagens são exibidas ao usuário sobre respectivos erros de conflitos *shift/reduce* e *reduce/reduce* bem como mensagens acerca de regras inalcançáveis e *tokens* não utilizados. Além do mais, é gerado um arquivo com as regras gramaticais e com as informações das tabelas de análise sintática para melhorar a performance do compilador removendo a necessidade deste gerar a tabela toda vez que ocorrer a análise sintática. A listagem 8 mostra parte dos dados do arquivo gerado para a gramática da *Flex-Pl*.

Listagem 8: Definições parciais da gramática da *Flex-Pl* e tabela parcial de análise sintática *LALR,* armazenadas no arquivo parsing.out gerado pelo *Ply*. Fonte: Acervo do autor.

```
Rule 0     S' -> program
Rule 1     program -> function_definition program
Rule 2     program -> class_definition program
Rule 3     program -> function_call
Rule 4     program -> empty
Rule 5     function_call -> ID LPAREN function_call_args RPAREN
Rule 6     function_call_args -> expression
Rule 7     function_call_args -> function_call
Rule 8     function_call_args -> empty
Rule 9      class_definition -> DEFINE CLASS ID LPAREN RPAREN LBRACE
program RBRACE
Rule 10        function_definition  ->  DEFINE  data_type  ID  LPAREN
function_definition_args RPAREN block
Rule 11    function_definition_args -> var_list
Rule 12    function_definition_args -> empty
Rule 13    var_list -> var_declaration COMMA var_list
Rule 14    var_list -> var_declaration
Rule 15    var_declaration -> data_type ID
Rule 16    data_type -> VOID
Rule 17    data_type -> INT
Rule 18    data_type -> FLOAT
Rule 19    data_type -> STRING
Rule 20    data_type -> CHAR
…
Parsing method: LALR


state 0


    (0) S' -> . program
    (1) program -> . function_definition program
    (2) program -> . class_definition program
    (3) program -> . function_call
    (4) program -> . empty
        (10)  function_definition  ->  .  DEFINE  data_type  ID  LPAREN
function_definition_args RPAREN block
        (78)  function_definition  ->  .  DEFINE  data_type  ID  LPAREN
```

```
function_definition_args RPAREN error
     (9) class_definition -> . DEFINE CLASS ID LPAREN RPAREN LBRACE
program RBRACE
    (5) function_call -> . ID LPAREN function_call_args RPAREN
    (76) empty -> .

    DEFINE          shift and go to state 7
    ID              shift and go to state 1
    $end            reduce using rule 76 (empty -> .)

    function_call                   shift and go to state 4
    empty                           shift and go to state 5
    function_definition             shift and go to state 6
    class_definition                shift and go to state 3
    program                         shift and go to state 2
…
```

O gerador tem a capacidade de resolver diversos problemas relativos a conflitos causados por ambiguidades, mesmo assim não é recomendado deixar a gramática ambígua porque isso pode trazer problemas indesejáveis posteriormente. Portanto uma vez que a gramática não é ambígua e/ou esses problemas puderam ser tratados e a análise de um código fonte ocorreu sem erros a próxima fase do compilador pode ocorrer, a fase de análise semântica.

5. Análise Semântica

Nessa parte os algoritmos implementados não trabalharam em conjunto com os algoritmos de análises léxica e sintática implementados manualmente, este foi implementado para usar a Árvore de Análise Sintática Abstrata gerada com auxílio do *Ply*. As Classes listadas na listagem 9 foram usadas nas funções que contém as definições gramaticais dispostas na listagem 7 como já explicado no tópico de análise sintática deste artigo, as quais são nodos da árvore, mantendo informações relevantes como nomes, valores, referencias para outros nodos e dados como número de linha e coluna onde uma determinada definição tenha ocorrido.

Listagem 9: Definições parciais dos nodos usados na criação da Árvore Sintática Abstrata. Fonte: Acervo do autor.

```
0 class Program():
1    def __init__(self, scope=None, anotherScope=None):
```

```
2         self.type = "Program"
3         self.scope = scope
4         self.anotherScope = anotherScope
5 # The scope of a function
6 class FunctionDefinition():
7       def __init__(self, dataType, functionName, arguments, scope,
line, column):
8          self.type = "Function Definition"
9          self.dataType = dataType
10         self.functionName = functionName
11         self.arguments = arguments
12         self.scope = scope
13         self.line = line
14         self.column = column
15
16 # A expression
17 class BinaryOp(CodeGenerator):
18      def __init__(self,left,op,right):
19         self.type = "Binop"
20         self.left = left
21         self.right = right
22         self.op = op
23
24 class Block:
25     def __init__(self, statements):
26         self.type = "New Scope. 'BLOCK'"
27         self.statements = statements
```

Obviamente nem todas as definições são dadas na listagem anterior, entretanto ao que está exposto possibilita observar o padrão que os nodos apresentam quanto a sua estrutura. A Figura 6 ilustra como a árvore fica disposta após está ser gerada com auxilio do analisador sintático do *Ply* após realizar a análise sintática do programa disposto na Listagem 10, sendo este um programa que calcula o quadrado de um número fornecido como argumento a função *square*.

Listagem 10: Algoritmo que computa o quadrado de um dado número n abstraído em Flex-Pl. Fonte: Acervo do autor.

```
0 define int square(int n) {
1       return n * n;
2 }
```

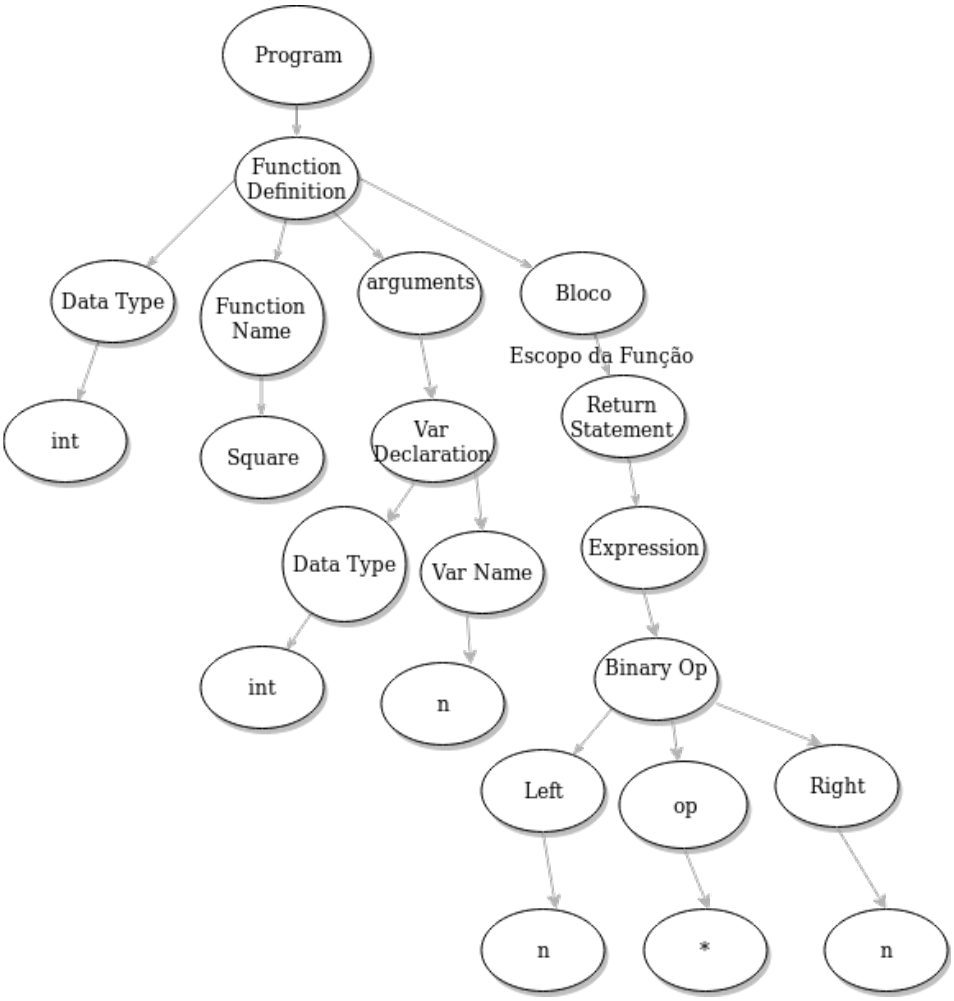

Figura 6: Árvore Sintática Abstrada que retrata o algoritmo da Listagem 10. Fonte: Acervo do autor.

Uma vez que a árvore é gerada, as informações relevantes são coletadas acessando-se os nodos da árvore em pré ordem e coletando as informações que são necessárias, como nome de funções, variáveis, tipos de dados, tipo de retorno da função tipos dos parâmetros formais e outros tipos de informações pertinentes. Para isso fora criada uma estrutura de dados análoga a tabela de símbolos, neste caso a estrutura criada foi uma tabela de símbolos com escopos a qual conta com métodos quais são usados para inserir escopos e realizar buscas por símbolos, e outras informações como o nível do escopo e um dicionário que contém valores os quais geralmente são funções declaradas no escopo global. Nessa estrutura geralmente são armazenadas apenas informações sobre as funções, ocorrendo-se isso através uma estrutura auxiliar a qual é um objeto

denominado de *Scope* (escopo) a qual armazena as informações pertinentes a respeito de um determinado escopo. Essas estruturas são mostradas na listagem 11.

Listagem 11: Estrutura de dados Tabela de símbolos com escopos. Fonte: Acervo do autor.

```
13 # Scope with function and Classes
14 class ScopedSymbolTable():
15     def __init__(self, name='None', instance=None, level=0):
16         self.globalScope = {}
17         self.name = name
18         self.instance = instance
19         self.level=level
20 
21     def insertScope(self, scope):
22         print('Insert: %s' % scope.name)
23         self.globalScope[scope.name] = scope
24 
25     def lookup(self, name):
26         #print('Lookup: %s. (Scope name: %s)' % (name, self.name))
27         #symbol is either an instance of the Symbol class or None
28         symbol = self.globalScope.get(name)
29         if symbol is not None:
30             return symbol
31 
32         return None
33 
34 class Scope():
35      def __init__(self, previousScope=None, name="Global", level=1,
instance=None):
36         self.previousScope = previousScope
37         self.level = level
38         self.name = name
39         self.instance = instance
40         self.scopes = []
41         self.symbols = {}
42 
43     def insertScope(self, scope):
44         self.scopes.append(scope)
```

```
45
46     def insertSymbol(self, symbol):
47         print('Insert: %s' % symbol.name)
48         self.symbols[symbol.name] = symbol
49
50    def lookup(self, name, currentScopeOnly=False):
51         #print('Lookup: %s. (Scope name: %s)' % (name, self.name))
52         #symbol is either an instance of the Symbol class or None
53         symbol = self.symbols.get(name)
54         if symbol is not None:
55             return symbol
56         if currentScopeOnly:
57             return None
58         # recursively go up the chain and lookup the name
59         if self.previousScope is not None:
60             return self.previousScope.lookup(name)
```

As abstrações *ScopedSymbolTable* e *Scope* contém algumas similaridades, ambas possuem atributos análogos e métodos para realizar inserções e buscas de símbolos, com a distinção da qual em *ScopedSymbolTable* são armazenadas informações sobre funções as quais podem ser recuperadas através do nome da função. Essas informações são estão contidas em uma instância da classe *Scope*, onde são armazenados nomes de variáveis seus tipos e outros escopos que possam existir como por exemplo o escopo de uma estrutura de controle *if/else* bem como laços como *for/while*. Além dessas informações, toda instância do tipo escopo tem uma referência para o escopo um nível acima desta. A Figura 7 ilustra o funcionamento dessas estruturas através de uma ilustração das estruturas *ScopedSymbolTable* e *Scope*.

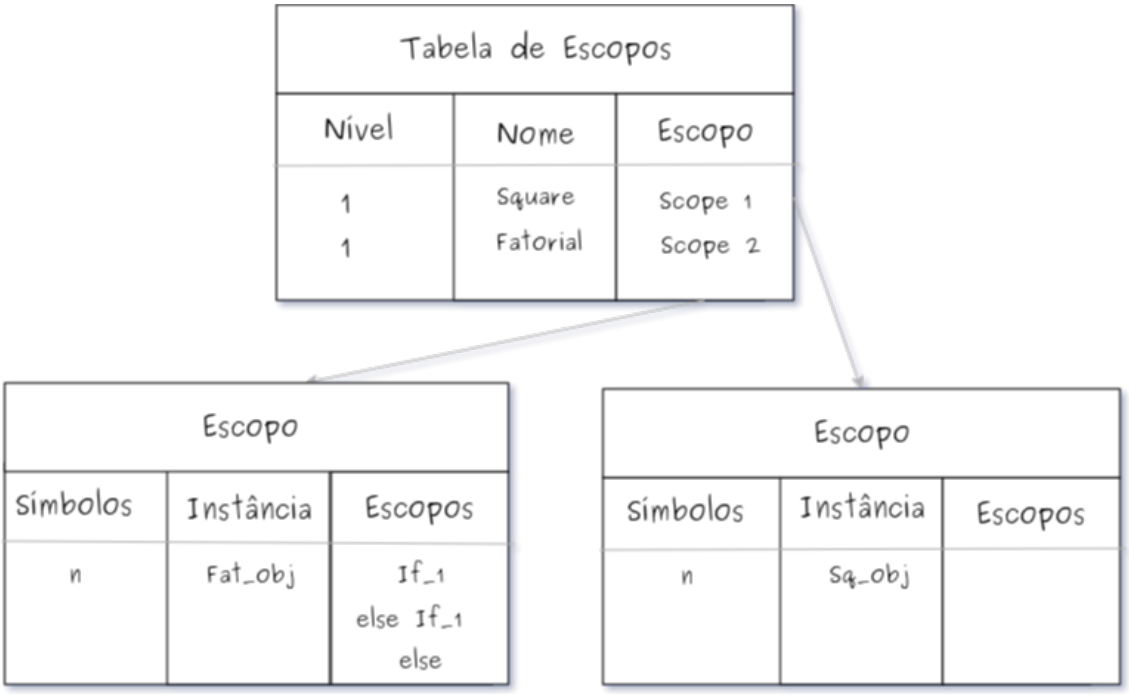

Figura 7: Ilustração da estrutura de dados Tabela de símbolos. Fonte: Adaptado de Ruslan's Blog. (Ruslan, 2017)

Uma classe denominada de *SemanticalAnalyzer* contém métodos que caminham recursivamente pela árvore sintática abstrata para coletar dados e utilizar-se das estruturas presentes na Listagem 11 e ilustradas na Figura 7, para salvar manter esses dados. Durante o decorrer dessa análise uma pilha é utilizada para manter os escopos, os quais são descartados quando não são mais necessários.

Sempre que uma definição é alcançada, como definições de funções ou declaração de variáveis por exemplo, o método *lookup* é invocado para verificar se o símbolo já fora definido/declarado anteriormente conforme mostra nas Listagens 12 e 13. Quando ocorre de o símbolo já existir, então um método de erro é acionado passando-se a mensagem de erro que este deve exibir e a instância do nodo que contêm as informações sobre o símbolo para que o método possa também mostrar qual é o símbolo e em qual linha e coluna ocorreu-se o erro e abortando-se a compilação. Quando o símbolo não existir, no caso de uma declaração de função um novo escopo é criado e é adicionado a tabela de escopos e seu escopo torna-se o escopo no topo da pilha. Quando há declaração de variáveis estas são adicionadas a tabela de símbolos, presente no escopo atual conforme mostra a Listagem 12.

Listagem 12: Analise semântica de declaração de variáveis. Fonte: Acervo do autor.

```
284 def __visitVarDeclaration__(self, varDeclaration):
285         currentScope = self.__getCurrentScope__()
286         if currentScope.lookup(varDeclaration.varName):
287                 print ("Error: Redefinition of the variable " + varDeclaration.varName +  " at Line: " + str(varDeclaration.line) + " column: " +  str(varDeclaration.column))
288         else:
289             varName = varDeclaration.varName
290             dtype = varDeclaration.dataType.dtype
291             instance = varDeclaration
292             symbol = Symbol(dtype, varName, instance)
293             currentScope.insertSymbol(symbol)
```

Listagem 13: Analise semântica de definição de funções. Fonte: Acervo do autor.

```
174    def __visitFunctionDefinition__(self, function):
175        # If the function already exists, then trow an error
176        try:
177            if self.symbolTable.lookup(function.functionName):
178                        self.errorProcedure("Error: Redefinition of function  " + function.functionName, function)
179        except:
180            exit()
181        else:
182            # If the function does not exist
183            # create a scope to it and push to
184            # the stack. Push the list of variables
185            # to the symbol table at the stack
186            functionName = function.functionName
187            currentScope = self.__getCurrentScope__()
188               newScope    =    Scope(currentScope,    functionName, instance=function)
189            block = function.scope
190            varList = function.arguments
191            self.symbolTable.insertScope(newScope)
192            self.stack.push(newScope)
193            # The variable list isn't empty
194            if varList:
195                self.__parser__(varList)
196            self.__parser__(block.statements)
```

```
197             if not self.stack.isEmpty():
198                 self.stack.pop()
```

Sempre que um símbolo é encontrado também busca-se pelo símbolo na tabela para ver se este já fora declarado anteriormente, e quando usado em expressões, sejam elas aritméticas ou lógicas, verifica-se se os tipos de dados concordam, por exemplo um inteiro pode ser usado em uma expressão que tenham objetos computacionais do tipo ponto flutuante desde que o lado esquerdo da atribuição seja do tipo *float* o contrário já não é permitido.

Além dessas outras verificações ocorrem conforme os nodos são alcançados até que todos sejam atingidos. Uma vez que isso ocorre, então o compilador passa para a faze de geração de código. No contexto da linguagem *Flex-Pl* código de montagem (*Assembly*) tendo como plataforma alvo *x86*.

6. Geração de Código Objeto

Uma vez que a análise ocorreu com sucesso, inicia-se a síntese. Para o compilador *Flex-Pl* escolheu-se a geração do código objeto em *Assembly* sem realizar otimizações ou geração de código intermediário de três endereços devido ao propósito educacional deste compilador, portanto não houve a necessidade de realizar tais fase uma vez que questões relativas a performance poderiam aumentar a complexidade deste.

Novamente a Árvore Sintática Abstrata é visitada de maneira similar a que ocorreu durante a fase de análise semântica, visitando-se cada nodo e realizando a geração de código durante a visitação dos nodos e salvando o código em um *buffer* intermediário.

Na geração de código usou-se um dicionário para manter o estado dos principais registradores de propósito geral de modo que o programa não permita que o valor em um registrador seja sobrescrito por outro valor. As posições que determinados valores encontram-se na *pilha* também são mantidos durante a geração do código em uma tabela de símbolos, sempre que uma variável é atingida é possível recuperar sua posição. A listagem 13, ilustra parte desse processo quando um nodo do tipo atribuição é alcançado.

Listagem 14: Método que gera código Assembly* para atribuições. *Fonte: Acervo do autor.

```
169 def __visitAssignment__(self, assignment):
171    #The left hand side of attribution has a variable declaration
172         if isinstance(assignment.var, VarDeclaration):
173            self.__visitVarDeclaration__(assignment.var)
174            if assignment.value is not None:
```

```
175                 r = self.__visitBinaryOp__(assignment.value)
176                                                     varPosition     =
self.getVarPosition(assignment.var.varName)
177                  self.data += "movl %" + r + "," + str(varPosition)
+ "(%ebp)\n"
178                 self.registers['eax'] = 0
```

Sempre que ocorre a aparição de variáveis a posição utiliza-se para tal variável na pilha é recuperada. No caso da atribuição o valor a ser atribuído é fica no registrador *eax* temporariamente, a posição da pilha é recuperada pelo nome da variável em uma tabela de símbolos, e por fim o valor é movido para tal posição.

Quando há definição de funções o próprio nome da função é usado como rótulo (*label*), já no caso de desvios de fluxo e laços utiliza-se um contador o qual seu valor é concatenado com nomes genéricos para realizar os saltos, conforme mostra nas Listagens 15 e 16.

Listagem 15: Método que gera código Assembly* para atribuições*. Fonte: Acervo do autor.

```
146    def __visitFunctionDefinition__(self, function):
147        functionName = function.functionName
148        self.data += ".type " + functionName + ", @function\n" +
functionName + ":\n\n"
149        self.data += "pushl %ebp\n"
150        self.data += "movl %esp, %ebp\n\n"
151        block = function.scope
152        varList = function.arguments
153        if varList:
154            self.__visitVarList__(varList, formalParameter=True)
155        self.__generator__(block.statements)
156        self.data += "movl %eax, %ebx" + "\n"
157        self.data += "movl %ebp, %esp" + "\n"
158        self.data += "popl %ebp" + "\n"
159        self.data += "ret" + "\n"
160        self.__resetParameterCounter__()
161        self.__resetRegisters__()
162        self.__resetLocalVars__()
163        self.__resetVarPosition__()
```

O nome da função é recuperado para ser usado como *label* e é armazenado no *buffer* temporario, o endereço do ponteiro base é adicionado ao topo da pilha e o ponteiro de pilha (*stack pointer*) passa a ser referenciado pelo ponteiro base. Quando tem se parâmetros formais código acerca destes é gerado através de uma chamada recursiva ao gerador de código. Em seguida é gerado o código do corpo da função e por fim os ponteiros são restaurados, o valor do retorno da função está no registrador *eax* e o controle é retornado para a função chamadora.

***Listagem 16: Método que gera código Assembly* para o laço *while*. Fonte: Acervo do autor.**

```
399     def __visitWhileStatement__(self, whileStatement):
400         startLabel =  self.instruction.startLoopLabel()
401         exitLabel  = self.instruction.exitLabel()
402     self.instruction.loopLabelCount += 1
403         self.data += startLabel + ":\n"
404         self.__visitBinaryOp__(whileStatement.logicalExpression)
405         self.instruction.labelCount += 1
406         self.__generator__(whileStatement.scope)
407         self.data += 'jmp ' +  startLabel + "\n"
408         self.data += exitLabel + ":\n"
```

O tipo de código gerado para um programa escrito na sintaxe da Flex-Pl dado na Listagem 17 o qual usa um laço *while* para o calculo do fatorial de um número de forma iterativa é exibido na Listagem 18.

Listagem 17: Algoritmo que computa o fatorial de um número iterativamente, abstraído na Flex-Pl. Fonte: Acervo do autor.

```
0 define int fat(int n) {
1        int result = 1;
2
3        while (n > 0) {
4                result = result * n;
5        n = n - 1;
6        }
7        return result;
8 }
```

Listagem 18: Código Assembly gerado sobre o algoritmo da Listagem 17. Fonte: Acervo do autor.

```
0 .type fat, @function
1 fat:
2 pushl %ebp
3 movl %esp, %ebp
4 subl $4, %esp
5 movl $1,%eax
6 movl %eax,-4(%ebp)
7 start_loop_0:
8 movl 8(%ebp), %eax
9 movl $0,%ebx
10 cmpl %ebx, %eax
11 jle exit_label_0
12 movl -4(%ebp), %eax
13 movl 8(%ebp), %ebx
14 imull %eax, %ebx
15 movl %ebx, %eax
16 movl %eax, -4(%ebp)
17 movl 8(%ebp), %ebx
18 movl $1,%ecx
19 subl %ecx, %ebx
20 movl %ebx, %ecx
21 movl %ebx, 8(%ebp)
22 jmp start_loop_0
23 exit_label_0:
24 movl -4(%ebp), %ecx
25 movl  %ecx,  %eax
26 movl %eax, %ebx
27 movl %ebp, %esp
28 popl %ebp
29 ret
```

Para a montagem do código gerado usou-se o montador *AS* (*Portable GNU Assembler*) para realizar a montagem do código, e para a ligação de bibliotecas do sistema operacional usou-se o ligador *LD* (*The GNU linker*), ambas as ferramentas estão

disponíveis na maioria das distribuições *GNU/Linux*. (BARTLETT, 2003)

7. Conclusão

A realização da construção de um compilador não é um simples processo para o estudante que está se deparando pela primeira vez com essa disciplina. O desenvolvimento de um compilador exige o conhecimento de diversas disciplinas da Ciência da Computação, em especial estruturas de dados e linguagens formais. Exigindo que o estudante tenha muitas vezes que recorrer a materiais de apoio que possam o ajudar e elucidar no momento da confecção de um compilador, quando existem lacunas no conhecimento do aluno e também durante o processo metodológico de desenvolvimento.

Durante o desenvolvimento do compilador para *Flex-PL*, diversas bibliografias e documentações foram consultadas servindo de apoio e elucidação para a criação deste. Sendo que estes materiais sempre foram de fácil acesso e sempre em grande número devido a maturidade dessa área. Durante o desenvolvimento deste, diversos algoritmos foram estudos e implementados, alguns dos quais pode-se citar devido sua importância histórica é o algoritmo de *Thompson* o qual converte uma expressão regular em um autômato finito não determinístico (Thompson, 1969), e o algoritmo de análise sintática LR (Knuth, 1965).

Devido a maturidade da área de compiladores, sua imensa vastidão de recursos bibliográficos, documentações e ferramentas, junto com a orientação acadêmica em sala de aula, tornou-se possível o sucesso desse projeto, o qual é de grande importância para a carreira de um Cientista da Computação devido a abrangência e aplicação em diversas áreas que as ferramentas estudadas na disciplina de compiladores proporcionam, tornando os conhecimentos adquiridos valiosos e estendendo o seu uso a outras disciplinas.

Mesmo com o sucesso do desenvolvimento deste, diversas dificuldades e desafios foram encontrados no decorrer do processo, mesmo com as ricas bibliografias disponíveis ainda assim, necessitou-se de muita calma e expertise no momento de implementar os algoritmos, houve-se bastante dificuldades para realizar a implementação de algoritmos geradores de analisadores léxicos e sintáticos devido a omissões e lacunas deixadas em algumas das bibliografias consultadas, também observou-se que referências para algoritmos geradores de analisadores léxicos a partir de expressões regulares são escassas e as referências existentes mostram-se ser de difícil compreensão devido a linguagem muito técnica/formal e lacuna de informações acerca da implementação destes. No decorrer do processo, a sintaxe da linguagem também sofreu algumas mudanças devido as gramáticas livres do contexto não terem poder suficiente para tratar determinadas construções idiomáticas que o projeto inicial da linguagem previa. Mesmo assim essas construções poderiam ser tratadas programaticamente ou através de outro recurso formal como uma máquina de Turing

por exemplo, entretanto isso aumentaria consideravelmente a complexidade do projeto bem como diminuiria a eficiência do compilador, fazendo com que a sintaxe da linguagem sofresse mudanças para que o projeto pudesse ter êxito. No entanto essas razões não atrapalharam no êxito deste projeto, devido não apenas essa área possuir uma grande gama de referências, mas também por possuir uma grande variedade de ferramentas que dão suporte a construção de compiladores.

As técnicas usadas na criação de compiladores, também são aplicadas fora do escopo de compiladores. Como por exemplo na leitura de dados estruturados, introdução a novos formatos de arquivos e problemas gerais de conversão de arquivos. Também muitos programas usam configurações ou especificações de arquivos que requerem um processamento que é muito similar a compilação. Dependendo da estrutura que certos dados possuem, é possível escrever uma gramática para analisá-los, usando para isso programas geradores de analisadores os quais podem ser gerados automaticamente através de ferramentas específicas. Tais técnicas também podem ser aplicadas na análise de arquivos no formato *HTML*, *PostScript* e em conversões de arquivos *TeX* para o formato *dvi* (Grune et al, 2012).

Mesmo sendo um compilador simples e de pequeno porte, este atingiu um nível de complexidade média com apenas 1567 linhas de código sem contar as linhas de código do gerador *Ply*. Em alguns momentos as implementações foram um pouco dificultosas, porém todas as fases foram implementas e testadas com sucesso, a geração do código *Assembly* ainda não está completa, alguns casos como expressões lógicas compostas ainda não podem ser geradas e o código para a função principal ainda é adicionado manualmente, mesmo assim o objetivo da disciplina foi atingido com sucesso e o compilador para a *Flex-Pl* continua a ser lapidado e futuras ocorreram conforme o desenvolvimento deste avança.

Referências

Abelson, Harold; Sussman, Gerald. J; Sussman, Julie. **Structure and interpretation of computer programs / Harold Abelson and Gerald Jay Sussman, with Julie Sussman.** – 2nd ed. p. com. – (Eletrical engineering and computer science series), 1996 The Massachusetts Institute of Technology. ISBN-13 978-0262-01153-2

Aho, Alfred V., et al. **Compiladores: princípios, técnicas e ferramentas**. 2. ed. São Paulo: Pearson Addison Wesley, 2008. x, 634 p. ISBN 9788588639249.

Bartlett, Jonathan. **Programing From Ground Up**. Editado por Dominick Bruno Jr. 2003. Distribuído através do website http://savannah.nongnu.org/projects/pgubook/.

Beazley, David. **PLY (Python Lex-Yacc)**. Disponível em: <http://www.dabeaz.com/ply/>. Acesso em 09/06/2018 às 22:00.

Gesser, C. E. **Gals Gerador de Analisadores Léxicos e Sintáticos**. Monografia

(Bacharel em Ciência da Computação) – Universidade Federal de Santa Catarina – UFSC Centro Tecnológico – CTC, Departamento de Informática e estatística – INE – Florianópolis - SC, Fevereiro de 2003.

Grune, Dick. *et al*. **Modern Compiler Design – Second Edition**. Springer Science + Business Media New York 2012. ISBN 978-1-4614-4699- 6

Hopcroft, John E.,; Ullman, Jeffrey D.; Motwani, Rajeev. **Introdução à teoria de autômatos, linguagens e computação**. 2. ed. Rio de Janeiro: Elsevier, 2003. 560 p. ISBN 9788535210729.

Knuth, Donald E. **On the Translation of Languages From Left to Right**. Revista – *Information and Control 8, 607-639 (1965).* Departamento de Matemática, Instituto de Technologia da California, Pasadena, California.

Levine, John R.; Mason, Tony; Brown, Doug (1992). **Lex & Yacc** (2 ed.). O'Reilly. pp. 1–2. ISBN 1-56592-000-7

Louden, Kenneth C. **Compiladores: princípios e prática**. 2. ed. São Paulo: Cengage Learning, 2004. xiv, 569 p. ISBN 9788522104222.

Price, Ana Maria de Alencar; Toscani, Simão Sirineo; UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL. **Implementação de linguagens de programação: compiladores**. 3. ed. Porto Alegre, RS: Sagra-DC Luzzatto, 2005. 195 p. (Livros didáticos (Universidade Federal do Rio Grande do Sul. Instituto de Informática); n.9). ISBN 8524106395 (broch.).

Sebesta, Robert W. **Conceitos de linguagens de programação**. 9. ed. Porto Alegre: Bookman, 2011. 792 p. ISBN 9788577807918.

Spivak, Ruslan. Ruslan's Blog. **Let's Build A Simple Interpreter**. Disponível em: <https://ruslanspivak.com/lsbasi-part1/>. Acesso em 09/06/2018 às 23:00.

Thompson, Ken. **Programming Tachniques: Regular Expression Search Algorithm**. Revista *Communications of The ACM* – Volume 11 Issue 6, Junho 1968, Páginas 419-422. New York, NY, USA.