Escola Politécnica da Universidade de São Paulo

Projeto de um compilador

Compilador para a linguagem Dartmouth BASIC

Autor:

Jhonata Antunes

18 de dezembro de 2018

SUMÁRIO

1 INTRODUÇÃO

Um compilador trata-se de um dos programas do software básico que desempenha como tarefa principal a conversão automática de programas-fonte (fornecidos como textos escritos em uma determinada linguagem de programação) em um texto equivalente cuja notação possa ser executada no computador (Neto, 2016).

1.1 Motivação

O avanço da tecnologia dos sistemas computacionais vem possibilitando a criação de grandes sistemas de software, capazes de realizar simulações físicas, cálculos complexos, executar modelos de redes neurais, processamento gráfico, entre outros. Para realizar tarefas complexas, é preciso abstrair as camadas mais baixas de hardware e software, de forma a concentrar o escopo de desenvolvimento apenas no problema alvo. Para que isso seja possível, é preciso utilizar linguagens de programação que trabalham no nível de abstração adequado ao projeto.

Para toda linguagem de programação, existe pelo menos um compilador ou interpretador correspondente. Devido ao fato da grande importância das linguagens de programação no ambiente de trabalho dos programadores, é fundamental que haja uma boa base de conhecimento a respeito dos conceitos sobre os compiladores, pois tal conhecimento pode ser útil em tarefas como detecção e correção de erros, otimização de desempenho, otimização de consumo de memória, entre outros.

1.2 Objetivo

O objetivo deste trabalho é projetar e implementar um compilador, que recebe como entrada a linguagem Dartmouth BASIC, e gera, como saída, código-objeto representado em linguagem simbólica.

Através deste trabalho será colocado em prática uma grande parte das teorias aprendidas ao longo da matéria de Linguagens e Compiladores.

1.3 Organização do trabalho

Este trabalho está dividido em cinco capítulos. O primeiro introduz os compiladores e os objetivos do trabalho. O segundo, contém a fase de projeto do compilador, onde os conceitos e estrutura de tópicos foram baseados em (Neto, 2016). O terceiro contém a descrição da fase de implementação. O quarto, apresenta os resultados obtidos, encarrando com a conclusão contida no quinto capitulo.

2 PROJETO DO COMPILADOR

2.1 Especificações gerais do compilador

Segundo (Neto, 2016), existe um conjunto de pontos da especificação de um compilador que destacam-se. Levando em consideração que o compilador deste trabalho tem finalidades didáticas, seguem os principais requisitos e os valores adotados para cada um deles:

- Finalidade a que se destina o compilador: trabalho acadêmico, com o objetivo de exercitar os conceitos teóricos sobre compiladores;
- Velocidade do compilador: qualquer;
- Velocidade do código-objeto: qualquer;
- Máquina hospedeira do compilador: qualquer;
- Máquina-alvo a ser utilizada pelo programa compilado: qualquer;
- Sistemas operacionais disponíveis em ambas as máquinas: qualquer;
- Ambiente de execução nativo disponível na máquina-alvo: qualquer;
- Grau de complexidade da linguagem-fonte: baixa;
- Número desejado de passos de compilação: qualquer;
- Compatibilidade no nível do programa-objeto: compatível com a máquina virtual MVN;
- Recursos computacionais necessários: nenhum, além do básico encontrado em computadores convencionais;
- Técnicas adotadas para a construção do compilador: arquitetura dirigida por sintaxe.

Vale notar que alguns requisitos são definidos como qualquer, devido ao fato de não ser relevante ao escopo ou não ter impacto no projeto.

2.2 Especificação da linguagem-fonte

Uma fase de grande importância no projeto de um compilador é a definição da linguagem a compilar, pois um simples erro de descrição da linguagem pode inutilizar o compilador.

A linguagem a compilar proposta neste trabalho é a linguagem Dartmouth BASIC, cuja gramática está formalizada a seguir, na notação de Wirth, onde, por motivo de simplicidade, foram omitidas as definições dos não terminais.

```
1  Program = BStatement { BStatement } int "END" .
2  BStatement = int ( Assign | Read | Data | Print | Goto | If |
       For | Next | Dim | Def | Gosub | Return | Remark ) .
3  Assign = "LET" Var "=" Exp .
4  Var = letter digit | letter [ "(" Exp { "," Exp } ")" ] .
5  Exp = { "+" | "-" } Eb { ( "+" | "-" | "*" | "/") Eb } .
6  Eb = "(" Exp ")" | Num | Var | ( "FN" letter | Predef ) "("
     Exp ")" .
7  Predef = "SIN" | "COS" | "TAN" | "ATN" | "EXP" | "ABS" | "LOG
     " | "SQR" | "INT" | "RND" .
8  Read = "READ" Var { "," Var } .
9  Data = "DATA" Snum { "," Snum } .
10 Print = "PRINT" [ Pitem { "," Pitem } [ "," ] ] .
11 Pitem = Exp | """ Character { Character } """ [ Exp ] .
12 Goto = ( "GOTO" | "GO" "TO" ) int .
13 If = "IF" Exp ( ">=" | ">" | "<>" | "<" | "<=" | "=" ) Exp "
     THEN" int .
14 For = "FOR" letter [ digit ] "=" Exp "TO" Exp [ "STEP" Exp ]
     .
15 Next = "NEXT" letter [ digit ] .
16 Dim = "DIM" letter "(" int { "," int } ")" { "," letter "("
     int { "," int } ")" } .
17 Def = "DEF" "FN" letter "(" letter [ digit ] ")" "=" Exp .
18 Gosub = "GOSUB" int .
```

```
19 Return = "RETURN" .
20 Remark = "REM" { Character } .
21 Int = digit { digit } .
22 Num = ( Int [ "." { digit } ] | "." Int ) [ "E" [ "+" | "-" ]
       Int ] .
23 Snum = [ "+" | "-" ] Num .
24 Character = letter | digit | special .
```

2.3 Projeto do analisador léxico

Através de uma simples análise na gramática da linguagem, é possível fazer um levantamento dos diferentes conjuntos de não terminais.

(a) Palavras reservadas:

END	LET	FN	SIN	COS	TAN
ATN	EXP	ABS	LOG	SQR	INT
RND	READ	DATA	PRINT	GOTO	IF
THEN	FOR	TO	STEP	NEXT	DIM
GOSUB	RETURN	REM			

(b) Átomos especiais:

identificadores	números inteiros

(c) Sinais de pontuação, operação e separação:

=	,	(	)	+	-	*	/	"	>	<

(d) Sinais compostos:

GO TO	DEF FN	>=	<=	<>

Portanto, esses são todos os *tokens* que devem ser extraídos pelo analisador léxico. Os *tokens* contém informação a respeito do tipo de átomo e informações complementares, de acordo com a Tabela 1.

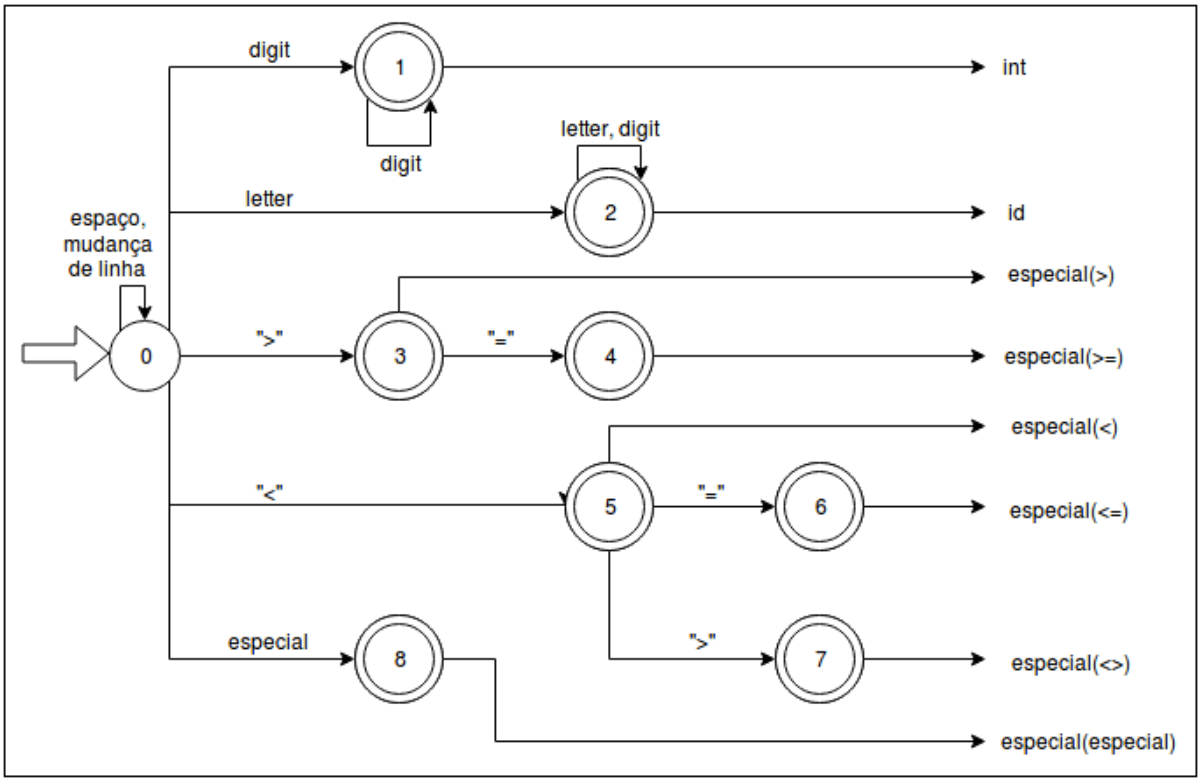
A partir das análises até aqui realizadas, é possível construir um autômato finito que será a base do transdutor que materializa o analisador léxico. A Figura 1 mostra uma

Tabela 1: Representação dos átomos (Neto, 2016)

Terminal	**Tipo do átomo**	**Informação complementar**
palavra reservada	a própria palavra reservada	irrelevante
identificador	<<identificador>>	índice do identificador na tabela de símbolos
número inteiro	<<número>>	valor numérico associado ao numeral encontrado
sinal	o próprio sinal	irrelevante
outro símbolo	o prórpio símbolo	irrelevante

representação do autômato do analisador léxico, onde, a rigor, os estados finais deveriam ser convertidos em não finais, adicionando-se transições em vazio para o estado inicial, com emissão de átomos, porém, foi omitido por motivo de simplicidade.

Figura 1: Transdutor léxico da linguem BASIC



Fonte: Autor

Dado o analisador léxico, foram realizados pequenos ajustes na gramatica, para que ela se enquadre, sem problemas, na fase de analise sintatica, uma vez que este analisador léxico não irá fornecer átomos dos tipos *letter*, *digit*, e sim, *id* e *int*, respectivamente. Essa mudança gerou um pequeno impacto na linguagem original, onde agora é possível definir variáveis com mais de dois caracteres. Assim sendo, a linguagem modificada ficou

da seguinte forma:

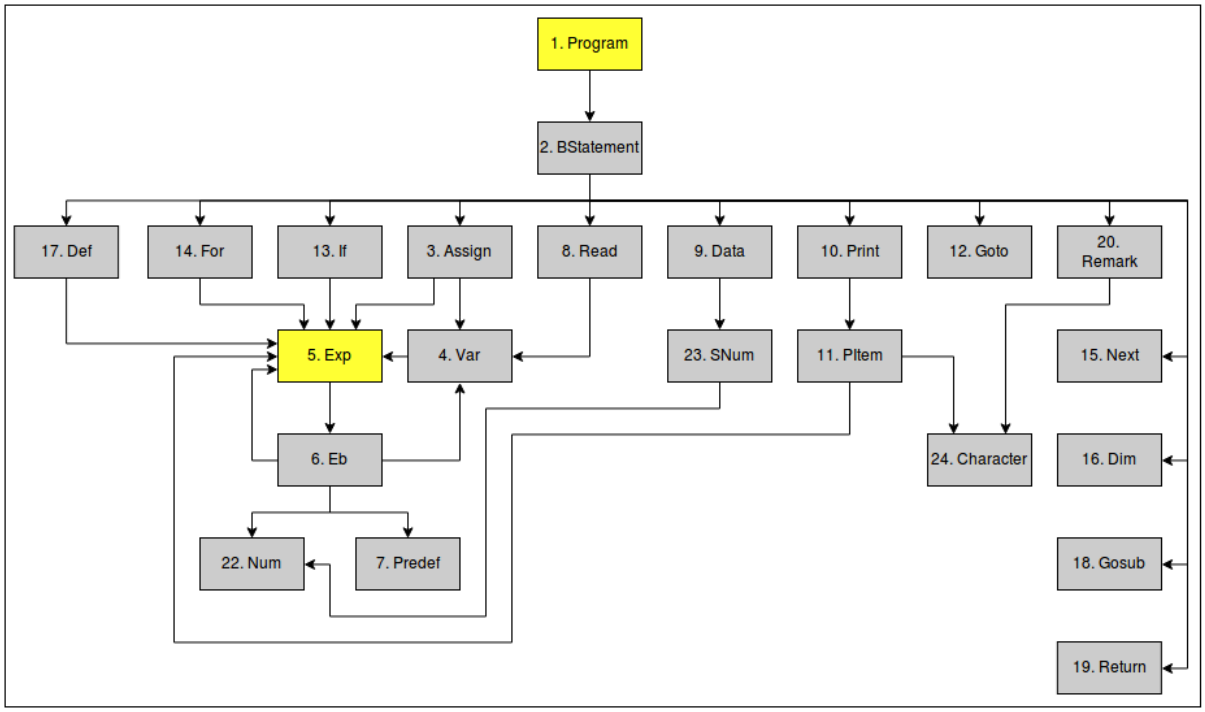
```
1  Program = BStatement { BStatement } int "END" .
2  BStatement = int ( Assign | Read | Data | Print | Goto | If |
       For | Next | Dim | Def | Gosub | Return | Remark ) .
3  Assign = "LET" Var "=" Exp .
4  Var = id [ "(" Exp { "," Exp } ")" ] .
5  Exp = { "+" | "-" } Eb { ( "+" | "-" | "*" | "/") Eb } .
6  Eb = "(" Exp ")" | Num | Var | ( "FN" id | Predef ) "(" Exp
       ")" .
7  Predef = "SIN" | "COS" | "TAN" | "ATN" | "EXP" | "ABS" | "LOG
       " | "SQR" | "INT" | "RND" .
8  Read = "READ" Var { "," Var } .
9  Data = "DATA" Snum { "," Snum } .
10 Print = "PRINT" [ Pitem { "," Pitem } [ "," ] ] .
11 Pitem = Exp | """ Character { Character } """ [ Exp ] .
12 Goto = ( "GOTO" | "GO" "TO" ) int .
13 If = "IF" Exp ( ">=" | ">" | "<>" | "<" | "<=" | "=" ) Exp "
       THEN" int .
14 For = "FOR" id "=" Exp "TO" Exp [ "STEP" Exp ] .
15 Next = "NEXT" id .
16 Dim = "DIM" id "(" int { "," int } ")" { "," id "(" int { ","
       int } ")" } .
17 Def = "DEF" "FN" id "(" id ")" "=" Exp .
18 Gosub = "GOSUB" int .
19 Return = "RETURN" .
20 Remark = "REM" { Character } .
21 Num = ( int [ "." int ] | "." int ) [ "E" [ "+" | "-" ] int ]
       .
22 Snum = [ "+" | "-" ] Num .
23 Character = id | int | special .
```

2.4 Projeto do analisador sintático

Para a determinação dos não terminais essenciais foi utilizada a árvore da gramática. A partir de uma simples analise da Figura 2, é possível notar que os não terminais

essenciais da gramática são *Program* e *Exp*.

Figura 2: Árvore da gramática da linguem BASIC, onde os não terminais essenciais estão destacados com a cor amarela.



Fonte: Autor

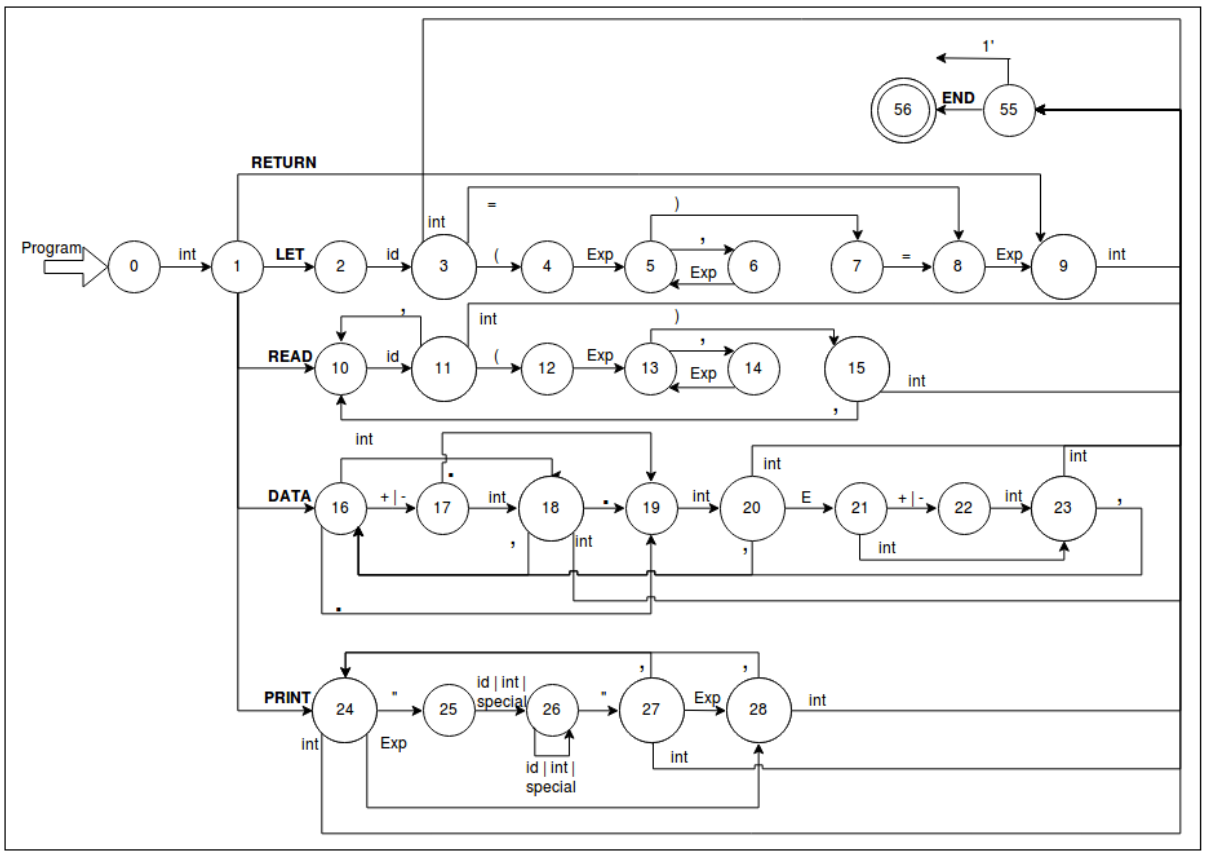
Para obter os autômatos do reconhecedor sintático é preciso realizar a eliminação das recursões desnecessárias. Uma vez que os não terminais essenciais já foram determinados, a gramática reduzida fica da seguinte maneira:

```
1 Program = ( int ( "LET" id [ "(" Exp { "," Exp } ")" ] "="
    Exp | "READ" (id [ "(" Exp { "," Exp } ")" ]) { "," id [
    "(" Exp { "," Exp } ")" ] } | "DATA" ([ "+" | "-" ] ( int
    [ "." int ] | "." int ) [ "E" [ "+" | "-" ] int ]) { ","
    ([ "+" | "-" ] ( int [ "." int ] | "." int  [ "E" [ "+" |
    "-" ] int ]) } | "PRINT" [ (Exp | """ (id | int | special)
     { (id | int | special) } """ [ Exp ]) { "," (Exp | """ (
    id | int | special) { (id | int | special) } """ [ Exp ])
    } [ "," ] ] | ( "GOTO" | "GO" "TO" ) int | "IF" Exp ( ">="
     | ">" | "<>" | "<" | "<=" | "=" ) Exp "THEN" int | "FOR"
    id "=" Exp "TO" Exp [ "STEP" Exp ] | "NEXT" id | "DIM" id
    "(" int { "," int } ")" { "," id "(" int { "," int } ")" }
     | "DEF FN" id "(" id ")" "=" Exp | "GOSUB" int | "RETURN"
```

```
    | "REM" { (id | int | special) } ) ) { int ( "LET" (id [
   "(" Exp { "," Exp } ")" ]) "=" Exp | "READ" (id [ "(" Exp
   { "," Exp } ")" ]) { "," id [ "(" Exp { "," Exp } ")" ] }
   | "DATA" ([ "+" | "-" ] ( int [ "." int ] | "." int ) [ "E
   " [ "+" | "-" ] int ]) { "," ([ "+" | "-" ] ( int [ "."
   int ] | "." int ) [ "E" [ "+" | "-" ] int ]) } | "PRINT" [
    (Exp | """ (id | int | special) { (id | int | special) }
   """ [ Exp ]) { "," (Exp | """ (id | int | special) { (id |
    int | special) } """ [ Exp ]) } [ "," ] ] | ( "GOTO" | "
   GO" "TO" ) int | "IF" Exp ( ">=" | ">" | "<>" | "<" | "<="
    | "=" ) Exp "THEN" int | "FOR" id "=" Exp "TO" Exp [ "
   STEP" Exp ] | "NEXT" id | "DIM" id "(" int { "," int } ")"
    { "," id "(" int { "," int } ")" } | "DEF FN" id "(" id
   ")" "=" Exp | "GOSUB" int | "RETURN" | "REM" { (id | int |
    special) } ) } int "END" .
2  Exp = { "+" | "-" } ( "(" Exp ")" | ( int [ "." int ] | "."
   int ) [ "E" [ "+" | "-" ] int ] | (id [ "(" Exp { "," Exp
   } ")" ]) | ( "FN" id | "SIN" | "COS" | "TAN" | "ATN" | "
   EXP" | "ABS" | "LOG" | "SQR" | "INT" | "RND" ) "(" Exp ")"
    ) { ( "+" | "-" | "*" | "/") ( "(" Exp ")" | ( int [ "."
   int ] | "." int ) [ "E" [ "+" | "-" ] int ] | id [ "(" Exp
    { "," Exp } ")" ] | ( "FN" id | "SIN" | "COS" | "TAN" | "
   ATN" | "EXP" | "ABS" | "LOG" | "SQR" | "INT" | "RND" ) "("
    Exp ")" ) } .
```

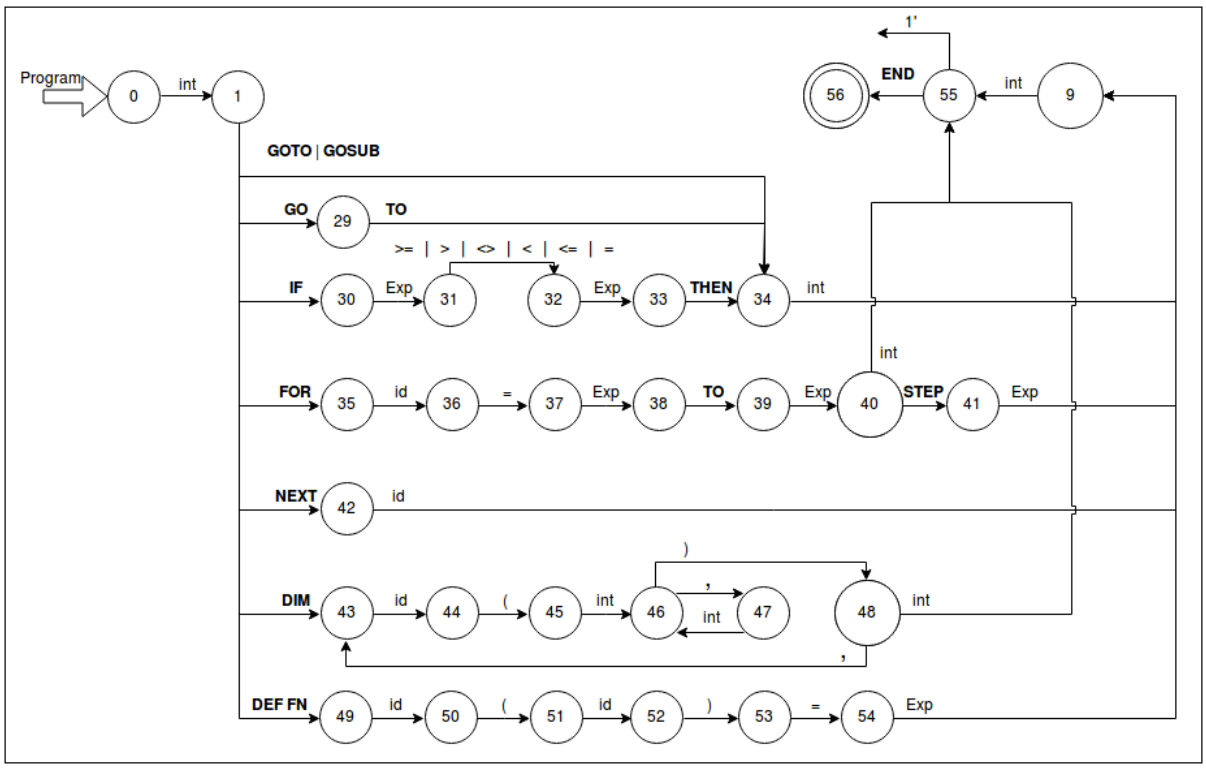
A partir da linguagem simplificada, foram construídos autômatos para todas as regras, como segue das Figuras 3 a 5. Vale notar que as aspas que envolem os terminais (incluindo as palavras reservadas) foram suprimidas.

Figura 3: Submáquina do comando *Program* (parte 1).



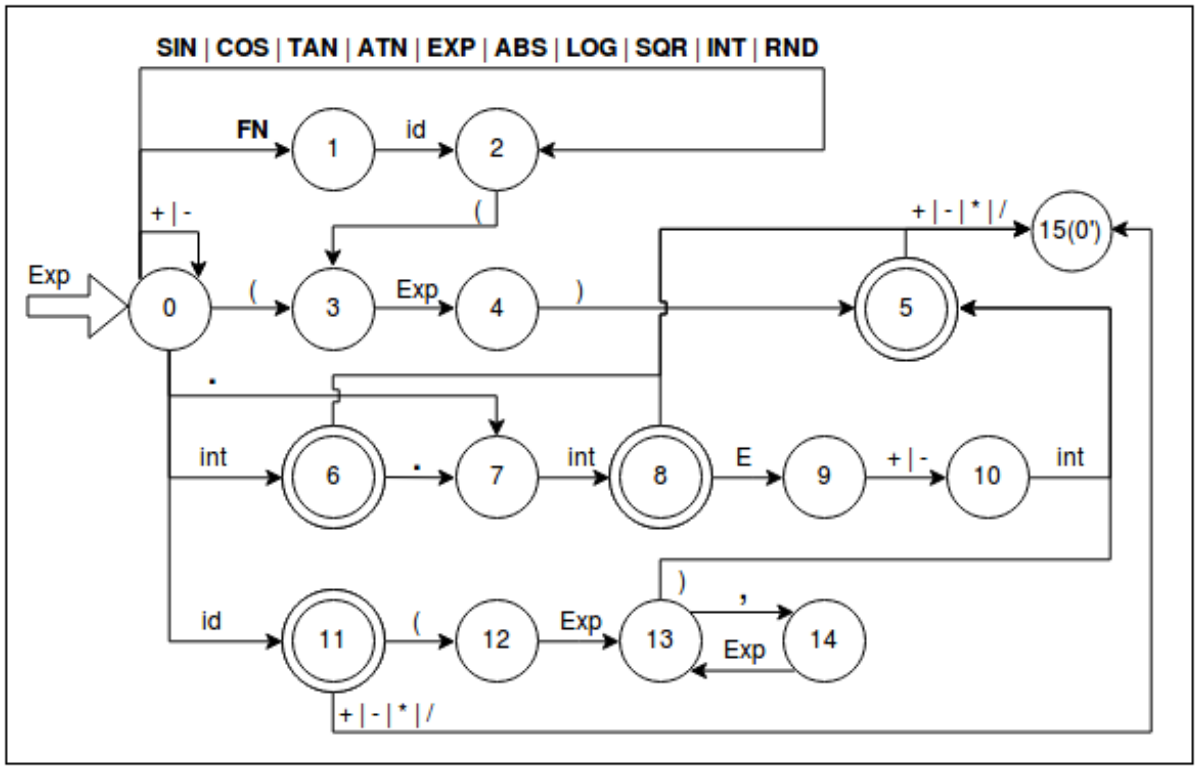
Fonte: Autor

Figura 4: Submáquina do comando *Program* (parte 2).



Fonte: Autor

Figura 5: Submáquina do comando *Exp*.



Fonte: Autor

3 IMPLEMENTAÇÃO

A implementação foi realizada na linguagem de alto nível Python, com a adição da biblioteca de testes Pytest. O cógido do projeto está disponível em (Antunes, 2018), assim como os arquivos de testes mencionados nas seções a seguir.

3.1 Sistema de arquivos

File system é o nome do módulo responsável por ler o arquivo a ser compilado. Ele recebe o nome de um arquivo, abre o arquivo e retorna uma linha, no formato binário, para cada evento de pedido de linha.

Para o teste automatizado deste módulo, foi criado um arquivo com três linhas, apenas para verificar se cada evento de leitura de linha era respondido adequadamente, ou seja, com uma linha de cada vez e, ao final, com um marcador de fim de arquivo.

3.2 Filtro ascii

Ascii filter é o nome do módulo responsável por converter caracteres binários para ascii. Este módulo gera um evento de solicitação de leitura de linha para o módulo *File system*, de forma a atender aos eventos de solicitação de caracteres ascii. A tarefa de conversão é realizada através de uma tabela que mapeia os valores binários aos respectivos caracteres ascii.

Para realizar o teste automatizado deste módulo, foi utilizado um arquivo com quatro linhas, contendo, em cada linhas, letras minúsculas, maiúsculas, caracteres especiais e dígitos. Foi feita a comparação dos caracteres convertidos pelo módulo com os equivalentes obtidos pela função *open*, de acesso a arquivos em Python.

3.3 Classificador ascii

Ascii classifier é o nome do módulo responsável por categorizar caracteres em uma das três categorias: digito, letra ou especial. Este módulo gera um evento de solicitação de carácter ascii ao módulo *Ascii filter*, classificando os caracteres recebidos toda vez que captura um evento de solicitação de carácter categorizado.

Para testar este módulo, foram utilizados dois arquivos, onde o primeiro contém uma série de caracteres diversos e o segundo, a categorização equivalente do arquivo de entrada, bastando apenas realizar uma simples comparação. A obtenção do arquivo equivalente foi feita de forma manual.

3.4 Classificador léxico

Lexical classifier é o nome do módulo responsável por gerar os *tokens* da linguagem. Este módulo gera uma série de eventos de solicitação de carácter ascii categorizado ao módulo *Ascii classifier*, percorrendo o autômato de reconhecimento léxico para gerar *tokens*, quando captura um evento de solicitação de *token*.

Para testar este módulo, foram utilizados dois arquivos, onde o primeiro contém uma série de declarações da linguagem e o segundo, os respectivos *tokens* do arquivo de entrada, bastando apenas realizar uma simples comparação. A obtenção do arquivo equivalente também foi feita de forma manual.

3.5 Reclassificador léxico

Lexical reclassifier é o nome do módulo responsável por realizar o pós processamento dos *tokens* da linguagem, de forma a recategorizar os *tokens* da classe *id* para a classe *key word* (palavra reservada), quando necessário. Este módulo gera um evento de solicitação de *token* ao módulo *Lexical classifier*, onde, com o auxílio de uma tabela com todas as palavras reservadas da linguagem, reclassifica, quando necessário, os *tokens*. Esté módulo atende ao evento de solicitação de *token* reclassificado.

O teste automatizado deste módulo é muito similar ao teste do *Lexical classifier*, onde a única diferença está no fato de o segundo arquivo possui a categoria de palavra reservada nos casos pertinentes.

3.6 Analisador sintático

Parser é o nome do módulo responsável realizar o reconhecimento sintático do código fonte. Sua implementação foi baseada nos dois não terminais essências da linguagem, *Program* e *Exp*. As chamadas de submáquinas foram implementadas com pilhas, onde a cada instanciação de uma nova submáquina, a anterior é empilhada, e a cada termino de submáquina, a anterior é desempilhada, para prosseguir com o reconhecimento.

Os testes foram realizados com o auxílio de três códigos da linguagem, sendo eles um código de teste com todas as declarações da linguagens, o algoritmo de ordenação *bubble sort* e o algoritmo de *fibonacci*.

Código 1:

```
10 LET I2 = 25+4
20 READ I3,J,K1
30 DATA 4,-5,0
40 PRINT "K1=", K1, "J=", J+2
50 GO TO 20
60 IF K1 < I3 THEN 10
70 FOR I=1 TO K1 STEP J
80 NEXT I
90 DIM A(3), B(2,5)
100 DEF FN F(X)= X*5+3
110 GOSUB 20
120 RETURN
140 END
```

Código 2 (*bubble sort*) (Fonte: (Shibata, 2018)):

```
10 DIM F(20)
12 FOR I = 0 TO 19
14 READ F(I)
20 NEXT I
30 DATA 3, 2, 1, 15, 17, 6, 0, 10, 11, 18, 4, 9, 19, 13, 16,
   14, 5, 8, 7, 12
40 LET S = 1
45 FOR I = 0 TO 18
50 IF F(I) <= F(I + 1) THEN 90
```

```
55 LET S = 0
60 LET T = F(I + 1)
70 LET F(I + 1) = F(I)
80 LET F(I) = T
90 NEXT I
110 IF S = 0 THEN 40
120 END
```

Código 3 (*fibonacci*) (Fonte: (Shibata, 2018)):

```
10 LET A = 0
20 LET B = 1
30 PRINT A
40 LET C = A + B
50 LET A = B
60 LET B = C
70 IF A < 10 THEN 30
80 END
```

4 RESULTADOS

Durante a implementação do reconhecedor sintático, foram abordadas algumas táticas que visavam a modularização os autômatos, de forma a se implementar o máximo possível individualmente, como submáquinas separadas. Esta abordagem permitiu uma rápida implementação, que, porém, continha problemas. Para corrigir alguns erros, foi necessário refatorar a gramática simplificada, de forma a eliminar a necessidade de realizar *backtracking* e alterar a implementação de algumas submáquinas para realizar *look ahead*. Por fim, alguns problemas, em relação aos autômatos, não puderam ser solucionados. Este processo resultou em grande desperdício de recursos (tempo, principalmente), pois muito retrabalho foi aplicado.

Após enfrentar estes problemas, foi decidido realizar a eliminação de todos os não terminais não essências, de forma a obter autômatos bem maiores e complexos, poŕem, um vez projetados, a implementação seria rápida e certeira.

Devido ao grande gasto do recurso tempo, não houve tempo para o projeto e implementação das ações semânticas, de forma que o resultado obtido foi um reconhecedor sintático. O reconhecedor sintático implementador foi aprovado nos três casos de testes descritos anteriormente.

5 CONCLUSÃO

Com a realização deste trabalho, foi possível colocar em prática uma parte da teoria de linguagens e compiladores. Os problemas encarados neste trabalho mostraram a importância da adoção de técnicas e métodos descritos na literatura, de forma a orientar o projeto de forma objetiva.

REFERÊNCIAS

1 NETO, J. J. *Introdução à compilação*. 2rd. ed. Rio de Janeiro, RJ, BRA: Elsevier, 2016.

2 ANTUNES, J. Github: basic-compiler. 2018. Disponível em: <https://github.com/jhonata-antunes/basic-compiler>. Acesso em: 17 dez. 2018.

3 SHIBATA, T. Testes em basic. 2018. Disponível em: <https://sites.google.com/view/tiagoshibata-pcs3848/1-testes-em-basic?authuser=0>. Acesso em: 17 dez. 2018.