

Criação de compilador para Robot-L: Compilador para Simulação de Robô Móvel

Hugo Henrique Oliveira, Krysthian Pires Lessa, Uálex Silva

¹Instituto de Matemática e Estatística – Departamento de Ciências da Computação
Universidade Federal do Bahia
(UFBA) Salvador – BA – Brazil

`hugoholiveira45@gmail.com, krysthianlessa@gmail, ualexsj@gmail`

Abstract. *This article aims to create a compiler for the Robot-L language. The Robot-L language is intended to provide some simple mechanisms for the operation and manipulation of mobile robots. The compiler made according to the compilation phases in the article will explain and exemplify all the process phases with the lexical parser, syntactic parser, semantic parser, and assembly language code generator, besides having usage examples, and images exemplifying each of the phases.*

Resumo. *Este artigo tem como finalidade a criação de um compilador, para a linguagem Robot-L, a linguagem Robot-L tem o propósito de providenciar alguns mecanismos simples para o funcionamento e manipulação de robôs moveis. O compilador feito de acordo com as fases de compilação no artigo iremos explicitar e exemplificar todas as fases do processo com o analisador léxico, analisador sintático, analisador semântico, e gerador de código em linguagem de montagem, além de ter exemplos de uso, e imagens exemplificando cada uma das fases.*

1. Introdução

Um compilador é um programa especial responsável pelo processamento de instruções escritas em uma linguagem de programação específica e então as transforma em linguagem de máquina ou "código" para que assim o processador de um computador possa usá-las, ou seja um compilador tem a função de realizar automaticamente, a tradução de textos, redigidos em uma determinada linguagem de programação, para algum outro formato que viabilize sua execução pelo computador [Johnson et al. 1975], o mesmo esta dividido entre algumas fases, são elas a Analise Léxica, Analise sintática, Analise Semântica, Geração de código e Otimização de código como podemos ver abaixo:

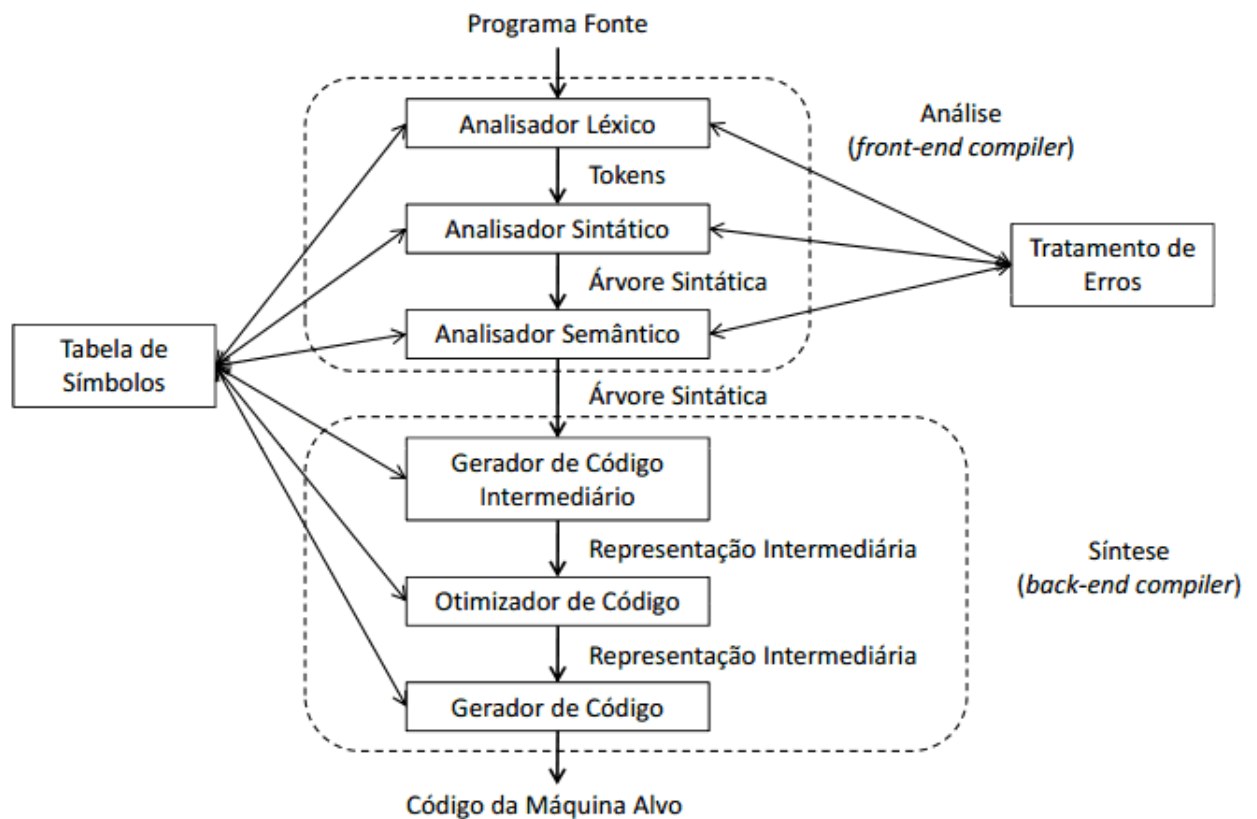


Figure 1. Fases do compilador

Ao longo do artigo iremos explicar qual a finalidade de cada fase do compilador, além de explicar como foi feita cada fase do compilador para a linguagem Robot-L proposta para o trabalho.

2. Informações gerais e definições

Linguagem:

3. Análise Léxica

Análise léxica é o processo de analisar a entrada de linhas de caracteres (tal como o código-fonte de um programa de computador) e produzir uma sequência de símbolos definidos como tokens.

3.1. Requisitos mínimos para o Léxico:

O analisador Léxico detém os seguintes requisitos:

1. Deve ler a sequência de caracteres que compõe o código fonte do programa, identificando-os e agrupando-os em uma sequência de tokens válidos da linguagem.
2. Deve ser capaz de identificar e reportar os erros léxicos encontrados no código fonte (e.g. símbolos desconhecidos ou identificador mal-formado). Para cada erro encontrado, deve-se informar o posicionamento (linha e coluna) no arquivo fonte de entrada em que o erro ocorreu.

3.2. Sintaxe básica

3.3. Tokens

Um Token em computação é um segmento de texto ou símbolo que pode ser manipulado por um analisador sintático, que fornece um significado ao texto. Os tokens definidos para essa gramática são:

1. Os terminais estão descritos entre aspas duplas e em negrito.
2. O símbolo * representa zero ou mais ocorrências do não-terminal à esquerda deste símbolo.
3. Produções opcionais estão entre colchetes.
4. O compilador deve aceitar palavras escritas em minúsculas ou maiúsculas.
5. Comentários são definidos em linhas iniciadas com o símbolo “”

3.4. Regras definidas na especificação

Temos aqui as regras que foram definidas na especificação da linguagem Robot-L, com as expressões regulares, palavras reservadas e comandos da linguagem, em base nessa especificação montamos a gramática que foi utilizada em todo o trabalho.

1. Programa ::= \programainicio" Declaracao* \execucaoinicio"
2. Comando \fimexecucao" \fimprograma"
3. Declaracao ::= \definainstrucao" identificador \como" Comando
4. Bloco ::= \inicio" Comando* "fim"
5. Comando ::= Bloco | Iteracao | Laco | Condicional | Instrução
6. Iteracao ::= \repita" Numero \vezes" Comando \fimrepita"
7. Laço ::= \enquanto" Condicao \faca" Comando \fimpara"
8. Condicional ::= \se" Condicao \entao" Comando \fimse" [\senao" Comando \fimsenao"]
9. Instrucao ::= \mova" Numero* [\passos"] | \Vire Para" Sentido |
10. Identificador | \Pare" | \Finalize" | \Apague Lampada"| \Acenda Lampada" | \Aguarde Ate" Condição
11. Condicao ::= \Robo Pronto" | \Robo Ocupado" | \Robo Parado" | \Robo Movimentando" | \Frente Robo Bloqueada" | \Direita Robo Bloqueada" | \Esquerda Robo Bloqueada" | \Lampada Acessa a Frente" | "Lampada Apagada a Frente" | \Lampada Acessa A Esquerda" | \Lampada Apagada A Esquerda" | \Lampada Acessa A Direita" | \Lampada Apagada A Direita"
12. Identificador ::= Letra(Letra|Digito)*
13. Numero ::= Digito*
14. Letra ::= \A" | \a" | \B" | \b" | ... | \z"
15. Digito ::= \0" | ... | \9"

16. Sentido ::= \esquerda" | \direita"

Com base nas regras propostas definimos então a seguinte gramática para a linguagem Robot-L:

```
1 programa' -> programa
2
3 programa -> programainicio declaracoes execucao fimprograma
4 programa -> programainicio execucao fimprograma
5
6 execucao -> execucaoinicio comando fimexecucao
7
8 declaracoes -> declaracao
9 declaracoes -> declaracao declaracoes
10
11 declaracao -> definainstrucao id como comando
12
13 comando -> bloco
14 comando -> iteracao
15 comando -> laco
16 comando -> condicional
17 comando -> instrucao
18
19 bloco -> inicio fim
20 bloco -> inicio pre_comando fim
21
22 pre_comando -> comando
23 pre_comando -> pre_comando comando
24
25 iteracao -> repita num vezes comando fimrepita
26
27 laco -> enquanto condicao faca comando fimpara
28
29 condicional -> se condicao entao comando fimse
30 condicional -> se condicao entao comando fimse senao comando fimsenao
31
32 instrucao -> mova
33 instrucao -> mova num
34 instrucao -> mova passos
35 instrucao -> mova num passos
36 instrucao -> vire_para sentido
37 instrucao -> id
38 instrucao -> pare
39 instrucao -> finalize
40 instrucao -> apague_lampada
41 instrucao -> acenda_lampada
42 instrucao -> aguarde_ate condicao
43
44 condicao -> robo_pronto
45 condicao -> robo_ocupado
46 condicao -> robo_parado
47 condicao -> robo_movimentando
48 condicao -> frente_robo_bloqueada
49 condicao -> direita_robo_bloqueada
50 condicao -> esquerda_robo_bloqueada
51 condicao -> lampada_acesa_a_frente
52 condicao -> lampada_apagada_a_frente
53 condicao -> lampada_acesa_a_esquerda
```

```

54 condicao -> lampada_apagada_a_esquerda
55 condicao -> lampada_acesa_a_direita
56 condicao -> lampada_apagada_a_direita
57
58 sentido -> esquerda
59 sentido -> direita

```

3.5. Expressões Regulares

digito = [0 - 9]

letra = [a-zA-Z]

numero = (digito)*

id = letra(letra — digito)*

3.6. Autômato finito determinísticos para a análise léxica

Os estados definidos para o autômato foi q0 como estado inicial, ID para constantes identificadoras, NUM para constantes numéricas, CMT para comentários e ACC para estado de aceitação, que é quando não ocorre erros léxicos na cadeia de entrada.

SEP = ' ', '/n', '/t'.

EOF = End of File (Fim de arquivo).

qqc = Qualquer caracter da tabela ASCII (exceto a quebra de linha).

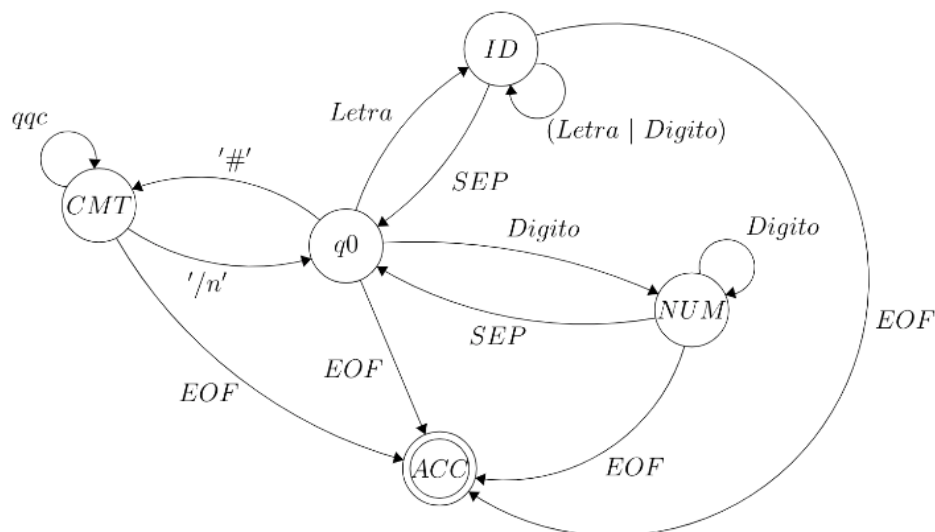


Figure 2. Autômatos finito determinístico para a análise léxica

3.7. Código fonte

Agora vamos apresentar e detalhar o código fonte gerado na primeira parte do trabalho, a análise léxica, feita na linguagem C++, optamos por publicar o código todo com as linhas

enumeradas, e logo abaixo explicar cada trecho de código por intervalo de linhas, para o melhor entendimento e compreensão na leitura e interpretação do mesmo.

```
1 #include <bits/stdc++.h>
2
3 using namespace std;
4
5 #define Q0 0
6 #define ID 1
7 #define NUM 2
8 #define CMT 3
9
10 typedef struct {
11     string str;
12 } Lexema;
13
14 typedef struct {
15     int estado;
16     int ult_linha;
17     int ult_coluna;
18     string str_acumulado;
19 } MaquinaDeEstados;
20
21 MaquinaDeEstados maquina;
22 vector<string> palavrasReservadas;
23 list<Lexema> listaDeLexemas;
24 int linha = 1;
25 int coluna = 1;
26 int qtdErros = 0;
27
28 void adicionaPalavrasReservadas() {
29     palavrasReservadas.push_back("programainicio");
30     palavrasReservadas.push_back("execucaoinicio");
31     palavrasReservadas.push_back("fimexecucao");
32     palavrasReservadas.push_back("fimprograma");
33     palavrasReservadas.push_back("definainstrucao");
34     palavrasReservadas.push_back("como");
35     palavrasReservadas.push_back("inicio");
36     palavrasReservadas.push_back("fim");
37     palavrasReservadas.push_back("repita");
38     palavrasReservadas.push_back("vezes");
39     palavrasReservadas.push_back("fimrepita");
40     palavrasReservadas.push_back("enquanto");
41     palavrasReservadas.push_back("faca");
42     palavrasReservadas.push_back("fimpara");
43     palavrasReservadas.push_back("se");
44     palavrasReservadas.push_back("entao");
45     palavrasReservadas.push_back("fimse");
46     palavrasReservadas.push_back("senao");
47     palavrasReservadas.push_back("fimsenao");
48     palavrasReservadas.push_back("mova");
49     palavrasReservadas.push_back("passos");
50     palavrasReservadas.push_back("vire");
51     palavrasReservadas.push_back("para");
52     palavrasReservadas.push_back("pare");
53     palavrasReservadas.push_back("finalize");
54     palavrasReservadas.push_back("apague");
55     palavrasReservadas.push_back("lampada");
56     palavrasReservadas.push_back("acenda");
```

```

57     palavrasReservadas.push_back("aguarde");
58     palavrasReservadas.push_back("ate");
59     palavrasReservadas.push_back("robo");
60     palavrasReservadas.push_back("pronto");
61     palavrasReservadas.push_back("ocupado");
62     palavrasReservadas.push_back("parado");
63     palavrasReservadas.push_back("movimentando");
64     palavrasReservadas.push_back("frente");
65     palavrasReservadas.push_back("direita");
66     palavrasReservadas.push_back("esquerda");
67     palavrasReservadas.push_back("acessa");
68     palavrasReservadas.push_back("a");
69     palavrasReservadas.push_back("apagada");
70 }
71
72 bool ehCharValido(char c) {
73     return (c == -1 || c == 9 || c == 10 || (c >= 32 && c <= 126));
74 }
75
76 bool eh_letra(char c) {
77     return ((c >= 65 && c <= 90) || (c >= 97 && c <= 122));
78 }
79
80 bool eh_numero(char c) {
81     return (c >= 48 && c <= 57);
82 }
83
84 bool eh_separador(char c) {
85     return ((c == 9) || (c == 10) || (c == 32));
86 }
87
88 bool eh_jogoDaVelha(char c) {
89     return (c == '#');
90 }
91
92 bool eh_EOF(char c){
93     return (c == -1);
94 }
95
96 void gerarLexema() {
97     Lexema lex;
98     lex.str = maquina.str_acumulado;
99     listaDeLexemas.push_back(lex);
100    maquina.str_acumulado = "";
101 }
102
103 void consome(char c) {
104     maquina.str_acumulado += c;
105     maquina.ult_linha = linha;
106     maquina.ult_coluna = coluna;
107 }
108
109 int qzero(char c) {
110     if (eh_letra(c)) {
111         consome(c);
112         return ID;
113     } else if (eh_numero(c)) {
114         consome(c);
115         return NUM;
116     } else if (eh_separador(c)) {
117         if (c == 10) {

```

```

118         linha++;
119         coluna = 0;
120     }
121     return Q0;
122 } else if (eh_jogoDaVelha(c)) {
123     consome(c);
124     return CMT;
125 } else if (eh_EOF(c)) {
126     return Q0;
127 } else {
128     cout << "Erro L xico " << linha << "." << coluna << "." << ": Caracter
129         inv lido." << endl;
130     qtdErros++;
131     return Q0;
132 }
133
134 int id(char c) {
135     if (eh_letra(c)) {
136         consome(c);
137         return ID;
138     } else if (eh_numero(c)) {
139         consome(c);
140         return ID;
141     } else if (eh_separador(c)) {
142         gerarLexema();
143         if (c == 10) {
144             linha++;
145             coluna = 0;
146         }
147         return Q0;
148     } else if (eh_EOF(c)) {
149         gerarLexema();
150         return Q0;
151     } else {
152         gerarLexema();
153         cout << "Erro L xico " << linha << "." << coluna << "." << ": Caracter
154             inv lido para constante identificadora." << endl;
155         qtdErros++;
156         return Q0;
157     }
158 }
159
160 int num(char c) {
161     if (eh_letra(c)) {
162         gerarLexema();
163         cout << "Erro L xico " << linha << "." << coluna << "." << ": N o
164             pode ter digito seguido de letra." << endl;
165         qtdErros++;
166         consome(c);
167         return Q0;
168     } else if (eh_numero(c)) {
169         consome(c);
170         return NUM;
171     } else if (eh_separador(c)) {
172         gerarLexema();
173         if (c == 10) {
174             linha++;
175             coluna = 0;
176         }
177         return Q0;

```



```

176     } else if (eh_EOF(c)) {
177         gerarLexema();
178         return Q0;
179     } else {
180         gerarLexema();
181         cout << "Erro L xico " << linha << "." << coluna << "." << ": Caracter
182             inv lido para constante num rica." << endl;
183         qtdErros++;
184         return Q0;
185     }
186 }
187
188 int cmt(char c) {
189     if(c == 10) {
190         gerarLexema();
191         linha++;
192         coluna = 0;
193         return Q0;
194     } else if(eh_EOF(c)) {
195         gerarLexema();
196         return Q0;
197     } else {
198         consome(c);
199         return CMT;
200     }
201 }
202
203 void analisadorLexico(string buffer) {
204     maquina.estado = Q0;
205
206     for(int i = 0; i < buffer.size(); i++, coluna++){
207
208         if (!ehCharValido(buffer[i]) && maquina.estado != CMT) {
209             cout << "Erro L xico " << linha << "." << coluna << "." << ":
210                 Carcter n o reconhecido pela linguagem." << endl;
211             qtdErros++;
212         } else {
213             switch (maquina.estado) {
214                 case Q0:
215                     maquina.estado = qzero(buffer[i]);
216                     break;
217                 case ID:
218                     maquina.estado = id(buffer[i]);
219                     break;
220                 case NUM:
221                     maquina.estado = num(buffer[i]);
222                     break;
223                 case CMT:
224                     maquina.estado = cmt(buffer[i]);
225                     break;
226                 default:
227                     cout << buffer[i] << endl;
228                     cout << "Tratar alguma coisa!\n";
229                     break;
230             }
231         }
232     }
233 }
234
235 int main()

```

```

235 {
236     string buffer;
237     char c;
238
239     while((c = getchar()) && (c != EOF))
240         buffer += c;
241     buffer += -1;
242
243     analisadorLexico(buffer);
244
245     if (qtdErros == 0)
246         cout << "Análise léxica concluída sem erros!" << endl;
247     else
248         cout << qtdErros << " erros léxicos encontrados." << endl;
249
250     return 0;
251 }

```

O compilador foi feito na linguagem C++ como podemos ver acima, inicialmente entra as linhas 1 e 26 criamos toda estrutura do compilador, definindo as variáveis estáticas, e a struct do compilador, em seguida até a linha 70, foi criada a função `adcionaPalavrasReservadas`, onde é adicionado a lista todas as palavras reservadas da linguagem, logo após até a linha 95 temos o tratamento para ser identificado a partir do carácter lido qual tipo de expressão ele será encaixado, então é verificado inicialmente se é um char válido e se faz parte da gramática, se é um número, letra, separador, hashtag, ou se é fim de arquivo, com isso feito, até a linha 200 é feita a lógica para análise léxica, seguindo a ordem do autômato gerado e a regra de cada expressão regular, seguindo as expressões regulares que foram demonstradas na sessão 3.5, assim que é encontrado algum erro Léxico em alguma parte da sequência de caracteres é então printado na tela "Erro Léxico linha x coluna y", seguido do tipo de erro léxico encontrado, que pode ser "Caracter não reconhecido pela linguagem.", "Caracter inválido para constante numérica.", "Não pode ter dígito seguido de letra.", "Caracter inválido para constante identificadora.", ": Caracter inválido.", ao final se não houver erro é mostrado a mensagem "Análise léxica concluída sem erros!", como pode ser visto na próxima sessão.

3.8. Retorno de erros Léxicos

Criamos arquivos de entrada, para o analisador léxico ser capaz de identificar e imprimir na tela os erros léxicos encontrados, abaixo segue 2 entradas (Teste1, Teste2) que não possuem erros léxicos, portanto o analisador deve imprimir na tela a string "Análise Léxica concluída sem erros!". E 3 entradas (Teste3, Teste4, Teste5) que possuem erros léxicos, a qual o analisador léxico deve imprimir o erro léxico encontrado, bem como a linha e coluna onde o erro ocorre.

1. Entradas sem erros Léxicos:

Teste1:

```

1 PROGRAMAINICIO
2     DEFINAiNSTRUCAO trilha COMO
3     INICIO
4         Mova 3 passos
5         Aguarde Ate Robo Pronto
6         Vire para ESQUERDA
7         Vire para DIREITA

```

```

8           Apagar LAMPADA
9           Mova 1 passo
10          Aguarde Ate Robo Pronto
11          FIM
12          EXECUCAOINICIO
13              Repita 3 VEZES Trilha
14              Vire Para Direita
15          FIMEXECUCAO
16 FIMPROGRAMA
17 robo pronto
18 ROBO PRONTO roBo oCuPADO OCUPADO ocupado          sd3443
19 fim
20 FIM          VEZES TRILHA ROBO
21 Aa A ATe ate
22 5656464564577346654745732365 gfdg456546 56456 dfg
23 varias VARIAS VE VEZES ave
24 PROGRAMAINICIO
25     DEFINAIN STRUCAO trilhaCOMO
26     INICIO
27         mova 3 passos
28         aguarde Ate Robo Pronto
29         Vire para ESQUERDA
30         Vire para DIREITA
31         Apagar LAMPADA
32         Mova 6 passo
33         Aguarde Ate Robo ProntoFIM
34     EXECUCAOINICIO
35         Repita 3 VEZES Trilha
36 vire Para Direita
37     FIMEXECUCAO
38 FIMPROGRAMA

```

Teste2:

```

1 adfsdfasdfas
2     DfsdfsdfsFUCAO trilhal23123 COMO
3     INICbdfdasdfsdCAO trilha COMO
4     IsdfsO
5         Msdfova 3 passos
6         Agusdfdrde Ate RsdfoBo PrsdF
7         Vsdfe psdfra ESQsfdFD
8         Vire para DSFREITA
9         Apa
10        Mova 3 passos
11        Aguarde Ate Robo Pronto
12        Vire para ESQUERDA
13        Aguarde Ate Robo Pronto
14        FIM
15        EXAAAAAAIO
16            Repita 3 VEZES Trilha123
17            Vire Pararde Ate Robo Pronto
18        FIM
19        EXEUJSDSDUICIO
20            Repita 3243324 VEZES Trilha
21            Vire Para Direita
22        FIdJJSDkAO
23 Aguarde Ate Robo Pronto
24            Vire par          Mova 3 passos
25            Aguarde Ate Robo Pronto
26            Vire para ESQUERDA
27            Vire para DIREITA

```

```

28         Apagar LAMPADA
29
30 EXECUCAAguarde Ate Robo Pronto
31         Vire par

```

2. Entradas com erros Léxicos:

Teste3:

```

1 PROGRAMAINICIO
2     DEFINAINSTRUCAO TRILH  COMO
3     INICIO
4         Mova 5zlo passos
5         Aguarde Ate Robo Pronto
6         Vire para ESQUERDA
7         Vire para DIREITA
8         Apagar LAMPADA
9         Mova 1 passo
10        Aguarde Ate Robo Pronto
11    FIM
12    EXECUCAOINICIO
13        Repita 7 VEZES Trilha
14        Vire Para Direita
15    FIMEXECUCAO
16 FIMPROGRAMA
17
18 #Teste de comentario com caracteres nao aceito fora de comentarios &*!@$?
19
20 Mesmo caracteres fora de comentario &*!@$?
21
22 PROGRAMAINICIO
23     DEFINAINSTRUCAO TRILHA COMO
24     INICIO
25         Mova 3 passos
26         Aguarde Ate Robo Pronto
27         Vire para ESQUERDA
28         Vire para DIREITA
29         Apagar LAMPADA
30         Mova 1* passo
31         Aguarde Ate Robo Pronto
32    FIM
33    EXECUCAOINICIO
34        Repita 3avc VEZES Trilha
35        Vire Para Direita
36    FIMEXECUCAO@gmail
37 FIMPROGRAMA

```

Teste4:

```

1         bo Pronto
2         Vire para ESQUERDA
3         Vire para DIREITA
4         Apagar LAMPADA
5         Mova 1%$ passo
6         Aguarde Ate Robo Pronto
7    FIM
8    EXECUCAOINICIO
9        Repita 3ABC VEZES Trilha
10       Vire Para letra123
11    FIMEXECUCAO
12 FIMPROGRAMA
13

```

```

14 _PRICI
15     DEFINAINSTRUCAO TRILHA COMO
16     INICIO
17         Mova 3 passos
18         Aguarde Ate Robo Pronto
19         Vire para %EQUERDA
20         Vire para DIREITA
21         Apagar LAMPADA
22         Mova 1 passo
23         Aguarde Ate Robo Pronto
24     FIM67&80
25     EXECUCAOINICIO
26         Repita 4 VEZES Trilha
27         Vire Para Direita
28     FIMEXECUCAO
29 FIMPROGRAMA
30
31 PROGRAMAINICIO
32     DEFINAINSTRUCAO TRILHA COMO
33     INICIO
34         Mova 3 passos
35         Aguarde Ate Robo Pronto
36         Vire para ESQUERDA
37         Vire para DIREITA
38         Apagar LAMPADA
39         Mova 1 passo
40         Aguarde Ate Robo Pronto
41     FIM
42     EXECUCAOINICIO
43         Repita 3 VEZES Trilha
44         Vire Para Direita
45     FIMEXECUCAO
46 FIMPROGRAMA

```

Teste 5:

```

1 PROGRAMAINICIO 78953 _9453 ASJASHJjashdjash 67655771231232$" $"56 67"% 6"$ "
   %7899 988234ijj fjaifjiai 92394234234234g
2     DEFINAINSTRUCAO trilha COMO      asfasf tirlha trilha prog mvmap17 (sad
   ) asdasd8()0 #asdsad sad"asdasd123"$&"&*"*(("&(*"(*%8854"*$*&"
   879689 dsaiu00
3 inicia INICIA inIcIa a_ _ --- _- _-$- 4b 4g b 5 gbb f5e 5g h65 fdg 6556S
4     INICIO PROGRAMAINICIO
5     DEFINAINSTRUCAO trilha COMO
6     INICIO PROGRAMAINICIO
7     DEFINAINSTRUCAO trilha COMO
8     INICIO
9     #asdasd*&*&%98*"*(("&89 89*&(& 68986 86*"(*"*( (& (*789& asdkoasdk
   897&(*&*( shad8ahj9d3h 9 &"%&*Yh983h49 8h*"*Y *("&*(G 93 *(&%"&*&
   G9DAS98D79 (*y Y 9 8 3      ^ ^#@@%"$"&*&
   hffgj jktr    Kjhkl    oii

```

NA figura 3, esta a saída da análise léxica executada pelo nosso analisador nas entradas Teste1, Teste2, Teste3 e Teste4:

```
krysthian@lessa: ~/MATA61-Compiladores/Analizador_Lexico
Arquivo Editar Ver Pesquisar Terminal Ajuda
krysthian@lessa:~/MATA61-Compiladores/Analizador_Lexico$ g++ lexico.cpp
krysthian@lessa:~/MATA61-Compiladores/Analizador_Lexico$ ./a.out < Testes/Corretos/teste01
Análise léxica concluída sem erros!
krysthian@lessa:~/MATA61-Compiladores/Analizador_Lexico$ ./a.out < Testes/Corretos/teste02
Análise léxica concluída sem erros!
krysthian@lessa:~/MATA61-Compiladores/Analizador_Lexico$ ./a.out < Testes/Errados/teste03
Erro Léxico 2.23.: Caractere não reconhecido pela linguagem.
Erro Léxico 2.24.: Caractere não reconhecido pela linguagem.
Erro Léxico 4.9.: Não pode ter dígito seguido de letra.
Erro Léxico 20.37.: Caractere inválido.
Erro Léxico 20.38.: Caractere inválido.
Erro Léxico 20.39.: Caractere inválido.
Erro Léxico 20.40.: Caractere inválido.
Erro Léxico 20.41.: Caractere inválido.
Erro Léxico 20.42.: Caractere inválido.
Erro Léxico 30.9.: Caractere inválido para constante numérica.
Erro Léxico 34.11.: Não pode ter dígito seguido de letra.
Erro Léxico 36.13.: Caractere inválido para constante identificadora.
12 erros léxicos encontrados.
krysthian@lessa:~/MATA61-Compiladores/Analizador_Lexico$ ./a.out < Testes/Errados/teste04
Erro Léxico 5.9.: Caractere inválido para constante numérica.
Erro Léxico 5.10.: Caractere inválido.
Erro Léxico 9.11.: Não pode ter dígito seguido de letra.
Erro Léxico 14.1.: Caractere inválido.
Erro Léxico 14.7.: Caractere não reconhecido pela linguagem.
Erro Léxico 14.8.: Caractere não reconhecido pela linguagem.
Erro Léxico 19.13.: Caractere inválido.
Erro Léxico 24.7.: Caractere inválido para constante identificadora.
8 erros léxicos encontrados.
krysthian@lessa:~/MATA61-Compiladores/Analizador_Lexico$
```

Figure 3. Execução de análise léxica

Já na figura 4, esta a saída da análise léxica executada pelo nosso analisador na entrada Teste5:

```
krysthian@lessa: ~/MATA61-Compiladores/Analizador_Lexico
Arquivo Editar Ver Pesquisar Terminal Ajuda
krysthian@lessa:~/MATA61-Compiladores/Analizador_Lexico$ ./a.out < Testes/Errados/teste05
Erro Léxico 1.22.: Caractere inválido.
Erro Léxico 1.59.: Caractere inválido para constante numérica.
Erro Léxico 1.60.: Caractere inválido.
Erro Léxico 1.61.: Caractere inválido.
Erro Léxico 1.62.: Caractere inválido.
Erro Léxico 1.68.: Caractere inválido para constante numérica.
Erro Léxico 1.69.: Caractere inválido.
Erro Léxico 1.72.: Caractere inválido para constante numérica.
Erro Léxico 1.73.: Caractere inválido.
Erro Léxico 1.75.: Caractere inválido.
Erro Léxico 1.76.: Caractere inválido.
Erro Léxico 1.88.: Não pode ter dígito seguido de letra.
Erro Léxico 1.116.: Não pode ter dígito seguido de letra.
Erro Léxico 2.65.: Caractere inválido.
Erro Léxico 2.69.: Caractere inválido para constante identificadora.
Erro Léxico 2.78.: Caractere inválido para constante identificadora.
Erro Léxico 2.79.: Caractere inválido.
Erro Léxico 3.23.: Caractere inválido para constante identificadora.
Erro Léxico 3.25.: Caractere inválido.
Erro Léxico 3.27.: Caractere inválido.
Erro Léxico 3.28.: Caractere inválido.
Erro Léxico 3.29.: Caractere inválido.
Erro Léxico 3.31.: Caractere inválido.
Erro Léxico 3.32.: Caractere inválido.
Erro Léxico 3.34.: Caractere inválido.
Erro Léxico 3.35.: Caractere inválido.
Erro Léxico 3.36.: Caractere inválido.
Erro Léxico 3.37.: Caractere inválido.
Erro Léxico 3.40.: Não pode ter dígito seguido de letra.
Erro Léxico 3.43.: Não pode ter dígito seguido de letra.
Erro Léxico 3.58.: Não pode ter dígito seguido de letra.
Erro Léxico 3.72.: Não pode ter dígito seguido de letra.
32 erros léxicos encontrados.
krysthian@lessa:~/MATA61-Compiladores/Analizador_Lexico$
```

Figure 4. Execução de análise léxica

É possível observar que nosso analisador é capaz de identificar o erro léxico, a linha e coluna onde ocorrem o erro, bem como o tipo de erro que ocorre e o total de erros léxicos identificados. Verificamos a análise léxica da entrada teste3 executada na figura 3, verificamos que a primeira string informa um erro léxico identificado na linha 2 e coluna 23, o tipo de erro impresso informa um caractere não reconhecido pela linguagem. Analisando a linha qual o erro se refere DEFINAINSTRUCAO TRILHá COMO, verificamos que o caractere á encontrado na coluna 23 não é um token válido.

4. Análise Sintática

A segunda fase do nosso compilador foi desenvolvida na análise sintática, nessa fase geramos a estrutura gramatical da gramática descrita na seção 3.4. A análise sintática transforma o texto de entrada em uma estrutura de dados, o que é conveniente para processamento posterior e capturar a hierarquia implícita desta entrada. Na análise léxica foi obtido um grupo de tokens, este grupo será usado nessa segunda fase para que o analisador sintático use um conjunto de regras para construir uma árvore sintática da estrutura.

A análise sintática pode ser realizada de duas maneiras:

Descendente (top-down) - um analisador pode iniciar com o símbolo inicial e tentar transformá-lo na entrada de dados. Intuitivamente, o analisador inicia dos maiores elementos e os quebra em elementos menores. Exemplo: analisador sintático LL.

Ascendente (bottom-up) - um analisador pode iniciar com uma entrada de dados e tentar reescrevê-la até o símbolo inicial. Intuitivamente, o analisador tenta localizar os elementos mais básicos, e então elementos maiores que contêm os elementos mais básicos, e assim por diante. Exemplo: analisador sintático SLR.

Tendo em vista que não há necessidade de implementar um analisador mais complexo como o LALR (quando há análise de símbolos posteriores), devido à simplicidade de implementação utilizamos o analisador sintático SLR na construção da nossa estrutura gramatical.

Uma gramática é SLR se for possível construir uma tabela SLR para ela. A construção da tabela SLR se baseia no conjunto canônico de itens LR(0)

4.1. Regras Sintáticas

Segue os requisitos básicos descritos para construção do analisador sintático:

1. Deve ser capaz de agrupar, hierarquicamente, a sequência de válidos em
2. frases gramaticais e representá-la através de árvores semânticas.
3. Deve validar se as sentenças (frases gramaticais) estão de acordo com a gramática especificada para a linguagem.
4. Deve estar apto para identificar e reportar os erros sintáticos encontrados no código fonte. Para cada erro encontrado, deve-se informar o posicionamento (linha) no arquivo fonte de entrada em que o erro ocorreu.

4.2. First e Follow da gramática

O conjunto follow é necessário na construção da tabela SLR, pois os analisadores de SLR usam o cálculo follow (A) para selecionar os símbolos de lookahead a serem esperados para cada não-terminal concluído. A figura 5 apresenta o cálculo de first e follow para a gramática Robot-L.

Nonterminal	FIRST	FOLLOW
programa	{programainicio}	{ \$ }
execucao	{execucaoinicio}	{fimprograma}
declaracoes	{definainstrucao}	{execucaoinicio}
declaracao	{definainstrucao}	{execucaoinicio,definainstrucao}
comando	{inicio,repita,enquanto,se,mova,vire_para,id,pare,finalize,apague_lampada,acenda_lampada,aguarde_ate}	{fimexecucao,execucaoinicio,definainstrucao,fim,inicio,repita,enquanto,se,mova,vire_para,id,pare,finalize,apague_lampada,acenda_lampada,aguarde_ate,fimrepita,fimpara,fimse,fimsenao}
bloco	{inicio}	{fimexecucao,execucaoinicio,definainstrucao,fim,inicio,repita,enquanto,se,mova,vire_para,id,pare,finalize,apague_lampada,acenda_lampada,aguarde_ate,fimrepita,fimpara,fimse,fimsenao}
pre_comando	{inicio,repita,enquanto,se,mova,vire_para,id,pare,finalize,apague_lampada,acenda_lampada,aguarde_ate}	{fim,inicio,repita,enquanto,se,mova,vire_para,id,pare,finalize,apague_lampada,acenda_lampada,aguarde_ate,fimrepita,fimpara,fimse,fimsenao}
iteracao	{repita}	{fimexecucao,execucaoinicio,definainstrucao,fim,inicio,repita,enquanto,se,mova,vire_para,id,pare,finalize,apague_lampada,acenda_lampada,aguarde_ate,fimrepita,fimpara,fimse,fimsenao}
laco	{enquanto}	{fimexecucao,execucaoinicio,definainstrucao,fim,inicio,repita,enquanto,se,mova,vire_para,id,pare,finalize,apague_lampada,acenda_lampada,aguarde_ate,fimrepita,fimpara,fimse,fimsenao}
condicional	{se}	{fimexecucao,execucaoinicio,definainstrucao,fim,inicio,repita,enquanto,se,mova,vire_para,id,pare,finalize,apague_lampada,acenda_lampada,aguarde_ate,fimrepita,fimpara,fimse,fimsenao}
instrucao	{mova,vire_para,id,pare,finalize,apague_lampada,acenda_lampada,aguarde_ate}	{fimexecucao,execucaoinicio,definainstrucao,fim,inicio,repita,enquanto,se,mova,vire_para,id,pare,finalize,apague_lampada,acenda_lampada,aguarde_ate,fimrepita,fimpara,fimse,fimsenao}
condicao	{robo_pronto,robo_ocupado,robo_parado,robo_movimentando,frente_robo_bloqueada,direita_robo_bloqueada,esquerda_robo_bloqueada,lampada_acessa_a_frente,lampada_apagada_a_frente,lampada_acessa_a_esquerda,lampada_apagada_a_esquerda,lampada_acessa_a_direita,lampada_apagada_a_direita}	{faca,entao,fimexecucao,execucaoinicio,definainstrucao,fim,inicio,repita,enquanto,se,mova,vire_para,id,pare,finalize,apague_lampada,acenda_lampada,aguarde_ate,fimrepita,fimpara,fimse,fimsenao}
sentido	{esquerda,direita}	{fimexecucao,execucaoinicio,definainstrucao,fim,inicio,repita,enquanto,se,mova,vire_para,id,pare,finalize,apague_lampada,acenda_lampada,aguarde_ate,fimrepita,fimpara,fimse,fimsenao}

Figure 5. First e Follow da gramática

4.3. Conjunto de Itens

Um item para uma gramática G é uma regra de produção com alguma indicação do que já foi derivado/consumido na regra durante a análise sintática, na figura 6 esta a demonstração da fase inicial de construção da nossa tabela de itens para a gramática definida na secção 3.4. A tabela completa está em anexo no arquivo sintático.csv para consulta.

Goto	Kernel	State	Closure
	{programa -> programainicio declaracoes execucao fimprograma}	0 {programa -> programainicio declaracoes execucao fimprograma}	
goto(0, programainicio)	{programa -> programainicio declaracoes execucao fimprograma}	1 {programa -> programainicio declaracoes execucao fimprograma; declaracoes -> declaracao; declaracoes -> declaracao declaracoes; declaracao -> definainstrucao id como comando}	
goto(1, declaracoes)	{programa -> programainicio declaracoes execucao fimprograma}	2 {programa -> programainicio declaracoes execucao fimprograma; execucao -> execucao inicio comando fim execucao}	
goto(1, declaracao)	{declaracoes -> declaracao; declaracoes -> declaracao declaracoes}	3 {declaracoes -> declaracao; declaracoes -> declaracao declaracoes; declaracoes -> declaracao; declaracoes -> declaracao declaracoes; declaracao -> definainstrucao id como comando}	
goto(1, definainstrucao)	{declaracao -> definainstrucao id como comando}	4 {declaracao -> definainstrucao id como comando}	
goto(2, execucao)	{programa -> programainicio declaracoes execucao fimprograma}	5 {programa -> programainicio declaracoes execucao fimprograma}	
goto(2, execucao inicio)	{execucao -> execucao inicio comando fim execucao}	6 {execucao -> execucao inicio comando fim execucao; comando -> bloco; comando -> iteracao; comando -> laco; comando -> condicional; comando -> instrucao; bloco -> inicio fim; bloco -> inicio pre_comando fim; iteracao -> repita num vezes comando fim repita; laco -> enquanto condicao faca comando fim para; condicional -> se condicao entao comando fim se; condicional -> se condicao entao comando fim se senao comando fim senao; instrucao -> move; instrucao -> move num; instrucao -> move num passos; instrucao -> move num passos; instrucao -> finalize; instrucao -> apague_lampada; instrucao -> acenda_lampada; instrucao -> aguarde_ate condicao}	
goto(3, declaracoes)	{declaracoes -> declaracao declaracoes}	7 {declaracoes -> declaracao declaracoes}	
goto(3, declaracao)	{declaracoes -> declaracao; declaracoes -> declaracao declaracoes}	3 {declaracoes -> declaracao; declaracoes -> declaracao declaracoes}	
goto(3, definainstrucao)	{declaracao -> definainstrucao id como comando}	4 {declaracao -> definainstrucao id como comando}	
goto(4, id)	{declaracao -> definainstrucao id como comando}	8 {declaracao -> definainstrucao id como comando}	
goto(5, fimprograma)	{programa -> programainicio declaracoes execucao fimprograma}	9 {programa -> programainicio declaracoes execucao fimprograma}	
goto(6, comando)	{execucao -> execucao inicio comando fim execucao}	10 {execucao -> execucao inicio comando fim execucao}	
goto(6, bloco)	{comando -> bloco}	11 {comando -> bloco}	

Figure 6. SLR.png

4.4. Tabela SLR

A tabela de ação é classificada por um estado do analisador sintático e um símbolo terminal (incluindo o terminal especial *que indica o final da entrada de dados*) e contém três tipos de ações :

mudança de estado (shift), escrito sn e indicando que o próximo estado é n redução (reduce), escrito rm e indicado que a redução com a regra gramatical m deve ser feita

aceitação (accept), escrito acc e indicando que o analisador sintático aceita a entrada de dados. A figura 7 esta a demonstração inicial da construção da tabela de ação do analisador SLR para a gramática definida na secção 3.4. A tabela completa está em anexo no arquivo sintático.csv para consulta.

	ACTION																															
	programa	fimpro	execuca	finexe	definain																											
State	início	grama	o início	cucao	strucao	id	como	início	fin	repita	num	vezes	finre	enquanto	faca	fimpara	se	entao	se	fin	senao	nao	finse	move	passos	vire	para	pare	finalize	apague	acenda	aguar
0	s1																															
1					s4																											
2			s6																													
3			r3		s4																											
4						s8																										
5		s9																														
6						s22	s16		s17				s18				s19						s20		s21	s23	s24	s25	s26	s27		
7			r4																													
8						s28																										
9																																
10				s29																												
11		r6	r6	r6	r6	r6	r6	r6	r6				r6	r6		r6	r6	r6	r6	r6	r6	r6	r6	r6	r6	r6	r6	r6	r6	r6	r6	
12		r7	r7	r7	r7	r7	r7	r7	r7				r7	r7		r7	r7	r7	r7	r7	r7	r7	r7	r7	r7	r7	r7	r7	r7	r7	r7	
13		r8	r8	r8	r8	r8	r8	r8	r8				r8	r8		r8	r8	r8	r8	r8	r8	r8	r8	r8	r8	r8	r8	r8	r8	r8	r8	
14		r9	r9	r9	r9	r9	r9	r9	r9				r9	r9		r9	r9	r9	r9	r9	r9	r9	r9	r9	r9	r9	r9	r9	r9	r9	r9	
15		r10	r10	r10	r10	r10	r10	r10	r10				r10	r10		r10	r10	r10	r10	r10	r10	r10	r10	r10	r10	r10	r10	r10	r10	r10	r10	
16						s22	s16	s30	s17				s18				s19						s20		s21	s23	s24	s25	s26	s27		
17										s33																						
18																																
19																																
20			r19	r19	r19	r19	r19	r19	r19	s49			r19	r19		r19	r19	r19	r19	r19	r19	r19	r19	s50	r19	r19	r19	r19	r19	r19	r19	
21																																
22		r24	r24	r24	r24	r24	r24	r24	r24				r24	r24		r24	r24	r24	r24	r24	r24	r24	r24	r24	r24	r24	r24	r24	r24	r24	r24	
23		r25	r25	r25	r25	r25	r25	r25	r25				r25	r25		r25	r25	r25	r25	r25	r25	r25	r25	r25	r25	r25	r25	r25	r25	r25	r25	
24		r26	r26	r26	r26	r26	r26	r26	r26				r26	r26		r26	r26	r26	r26	r26	r26	r26	r26	r26	r26	r26	r26	r26	r26	r26	r26	
25		r27	r27	r27	r27	r27	r27	r27	r27				r27	r27		r27	r27	r27	r27	r27	r27	r27	r27	r27	r27	r27	r27	r27	r27	r27	r27	
26		r28	r28	r28	r28	r28	r28	r28	r28				r28	r28		r28	r28	r28	r28	r28	r28	r28	r28	r28	r28	r28	r28	r28	r28	r28	r28	
27																																
28						s22	s16		s17				s18				s19						s20		s21	s23	s24	s25	s26	s27		
29		r2																														
30		r11	r11	r11	r11	r11	r11	r11	r11				r11	r11		r11	r11	r11	r11	r11	r11	r11	r11		r11	r11	r11	r11	r11	r11	r11	
31						s22	s16	s56	s17				s18				s19						s20		s21	s23	s24	s25	s26	s27		

Figure 7. LR.png

4.4.1. Código fonte

Abaixo segue o código fonte do nosso analisador sintático, em seguida vamos detalhar os trechos do código fonte que foi construído baseado na tabela de estado SLR apresentada anteriormente.

```

1 bool ehPalavraReservada(string lexema) {
2     for (int i = 0; i < palavrasReservadas.size(); i++)
3         if (palavrasReservadas[i] == lexema)
4             return true;
5
6     return false;
7 }
8
9 list<Lexema> pegarIdeNum(list<Lexema> lexemas) {
10     list<Lexema> :: iterator it;
11     Lexema lex;
12
13     for(it = lexemas.begin(); it != lexemas.end(); ++it) {
14         lex = *it;
15         if (!ehPalavraReservada(lex.str)) { // id ou Num
16             if (eh_numero(lex.str[0])) {
17                 lex.original = lex.str;
18                 lex.str = "num";
19                 *it = lex;
20             } else {
21                 lex.original = lex.str;
22                 lex.str = "id";
23                 *it = lex;
24             }
25         }
26     }
27 }

```

```

26     }
27
28     return lexemas;
29 }
30
31 string getEstadoNonTerminal(int estado, string str) {
32     for (int coluna = 46; coluna < COLUNAS; coluna++)
33         if (str == tableLR[0][coluna])
34             return tableLR[estado + 1][coluna];
35     return "false";
36 }
37
38 string getAcao(string str, int estado) {
39     for (int col = 1; col <= 45; col++)
40         if (str == tableLR[0][col])
41             return tableLR[estado][col];
42
43     return "false";
44 }
45
46 bool analisadorSintatico(list<Lexema> entrada) {
47     stack<string> pilha;
48     string acao;
49     int estado = 0;
50     Lexema lexema;
51
52     lexema.str = "$";
53     entrada.push_back(lexema);
54     pilha.push("$");
55     pilha.push("0");
56
57     while(!pilha.empty() && !entrada.empty()) {
58         lexema = entrada.front();
59         acao = getAcao(lexema.str, estado + 1);
60
61         if (acao == "acc")
62             return true;
63         if ((acao == "false") || (acao == "#")) {
64             cout << "Erro sint tico " << lexema.linha << "." << lexema.coluna
65                 << endl;
66             return false;
67         }
68
69         if (acao[0] == 's') {
70             pilha.push(lexema.str);
71             pilha.push(acao.substr(1));
72             entrada.pop_front();
73             estado = stoi(acao.substr(1));
74         }
75
76         if (acao[0] == 'r') {
77             int idcRegra = stoi(acao.substr(1));
78             int numDePops = (qtdRegras[idcRegra] * 2);
79             for (int i = 0; i < numDePops; i++)
80                 pilha.pop();
81             estado = stoi(pilha.top());
82             pilha.push(gramatica[idcRegra]);
83             string str = getEstadoNonTerminal(estado, gramatica[idcRegra]);
84             if (str == "acc")
85                 return true;
86             if (str == "false" || str == "#") {

```

```

86         cout << "Erro sint tico " << lexema.linha << "." << lexema.
            coluna << endl;
87         return false;
88     }
89     estado = stoi(str);
90     pilha.push(str);
91 }
92 }
93
94 return true;
95 }

```

Na nossa análise sintática criamos uma matriz que representa nossa tabela LR, e utilizamos uma estrutura de vetor para armazenar nossos não terminais. Antes de iniciar a análise sintática percorremos por todos os lexemas e adicionamos tokens para id e números, pois nossa gramática abstrai ambos. Depois implementamos o parse SLR que consiste em buscar na tabela uma ação a ser realizada, se esta for empilhar, empilha-se o terminal juntamente com o estado. Já se for uma redução, desempilhamos os terminais e empilhamos o terminal ao que este é reduzido, juntamente com o estado. Se em algum momento da busca na tabela, não encontramos uma ação ou um terminal correspondente na tabela, é porque o código possui erro sintático, imprimindo o local do erro retornando assim um valor falso para a função principal do programa. Mas se a execução for até o final, a busca na tabela encontrará o estado de aceitação, no qual retornamos um valor verdadeiro para a função principal. Com este algoritmo não conseguimos imprimir todos os erros sintáticos de uma só vez, por isso imprimimos o primeiro erro sintático e encerramos a análise sintática. Se este erro for corrigido e o programa executado novamente, aponta-se o próximo erro sintático, se houver, até que não haja mais erro sintático no código.

4.4.2. Erros sintáticos

Definimos entradas para o analisador sintático identificar erros sintáticos da linguagem, abaixo segue 3 entradas (Teste1, Teste2 e Teste3) que não possuem erros sintáticos, portanto o analisador deve imprimir na tela a string "Análise Sintática concluída sem erros!". E 3 entradas (Teste4, Teste5, Teste6) que possuem erros sintáticos, a qual o analisador deve imprimir a linha e coluna onde o erro ocorre.

1. Entradas sem erros sintáticos:

Teste1:

```

1 programainicio
2     definainstrucao compiladores como
3     inicio
4         vire para esquerda
5         acenda lampada
6         vire para direita
7     fim
8     execucaoinicio
9     inicio
10        repita 3 vezes compiladores fimrepita
11        vire para direita
12        se lampada acesa a frente entao
13            apague lampada

```

```

14         fimse senao
15             acenda lampada
16         fimsenao
17     fim
18     fimexecucao
19 fimprograma

```

Teste2:

```

1 programainicio
2     definainstrucao instrul0 como
3     inicio
4         enquanto frente robo bloqueada faca
5             inicio
6                 vire para direita
7                 mova 2 passos
8                 aguarde ate robo pronto
9                 vire para esquerda
10                mova 3 passos
11            fim
12        fimpara
13    fim
14    execucaoinicio
15    inicio
16        repita 3 vezes instrul0 fimrepita
17        vire para direita
18        se lampada acesa a frente entao
19            apague lampada
20        fimse senao
21            acenda lampada
22        fimsenao
23    fim
24    fimexecucao
25 fimprograma

```

Teste3

```

1 programainicio
2     execucaoinicio
3     inicio
4         vire para direita
5         se lampada acesa a frente entao
6             apague lampada
7         fimse senao
8             acenda lampada
9         fimsenao
10        enquanto direita robo bloqueada faca
11            inicio
12                mova 5 passos
13                aguarde ate robo pronto
14                vire para direita
15            fim
16        fimpara
17    fim
18    fimexecucao
19 fimprograma

```

2. Entradas com erros Sintáicos:

Teste4:

```

1 programainicio

```

```

2      definainstrucao instru10 como
3      inicio
4          enquanto frente robo bloqueada faca
5              inicio
6                  vire para direita
7                  mova 2 passos
8                  aguarde ate robo pronto
9                  vire para esquerda
10                 mova 3 passos
11             fim
12         fimpara
13     fim
14 fimprograma

```

Teste5:

```

1      programainicio
2      definainstrucao instru10 como
3      inicio
4          enquanto frente robo bloqueada faca
5              vire para direita
6              mova 2 passos
7              aguarde ate robo pronto
8              vire para esquerda
9              mova 3 passos
10         fimpara
11     fim
12     execucaoinicio
13         repita 3 vezes instru10 fimrepita
14         vire para direita
15         se lampada acesa a frente entao
16             apague lampada
17         fimse senao
18             acenda lampada
19         fimsenao
20     fimexecucao
21 fimprograma

```

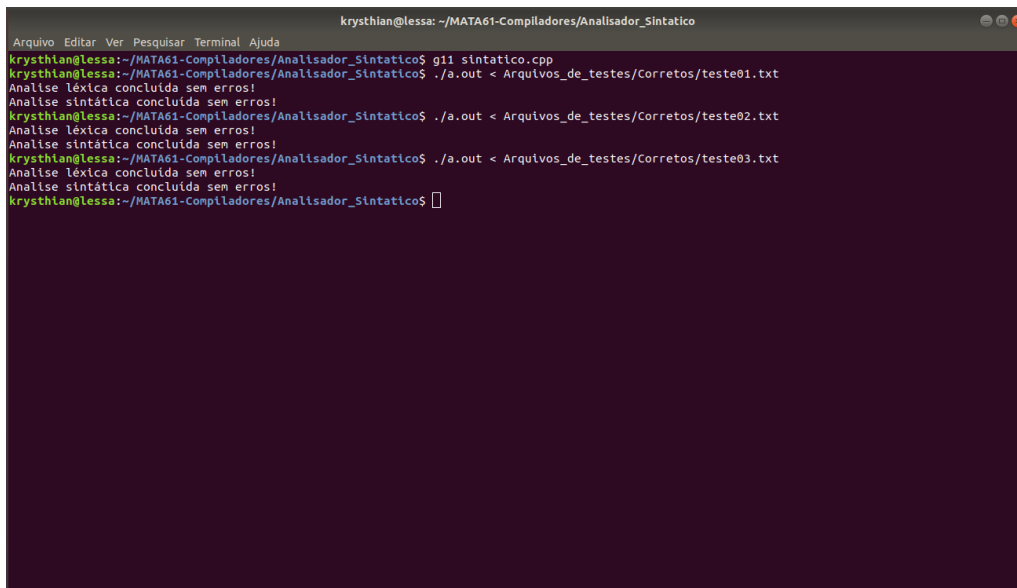
Teste 6:

```

1 programainicio
2     execucaoinicio
3         inicio
4             repita 3 vezes instru10 fimrepita
5             vire para direita
6             se lampada acesa a frente entao
7                 apague lampada
8             senao
9                 acenda lampada
10            enquanto direita robo bloqueada faca
11                inicio
12                    mova 5 passos
13                    aguarde ate robo pronto
14                    vire para direita
15                fim
16        fim
17    fimexecucao
18 fimprograma

```

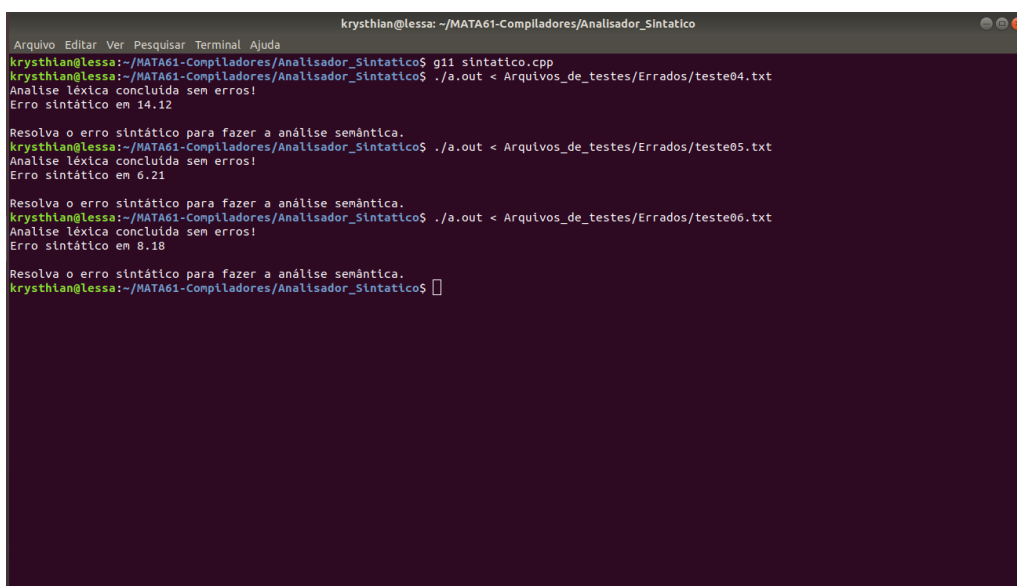
NA figura 8, esta a saída da análise sintática executada pelo nosso analisador nas entradas Teste1, Teste2, Teste3:



```
krysthian@lessa: ~/MATA61-Compiladores/Analizador_Sintatico
Arquivo Editar Ver Pesquisar Terminal Ajuda
krysthian@lessa:~/MATA61-Compiladores/Analizador_Sintatico$ g++ sintatico.cpp
krysthian@lessa:~/MATA61-Compiladores/Analizador_Sintatico$ ./a.out < Arquivos_de_testes/Corretos/teste01.txt
Analise léxica concluída sem erros!
Analise sintática concluída sem erros!
krysthian@lessa:~/MATA61-Compiladores/Analizador_Sintatico$ ./a.out < Arquivos_de_testes/Corretos/teste02.txt
Analise léxica concluída sem erros!
Analise sintática concluída sem erros!
krysthian@lessa:~/MATA61-Compiladores/Analizador_Sintatico$ ./a.out < Arquivos_de_testes/Corretos/teste03.txt
Analise léxica concluída sem erros!
Analise sintática concluída sem erros!
krysthian@lessa:~/MATA61-Compiladores/Analizador_Sintatico$
```

Figure 8. Execução de analisador sintático

Já na figura 9, esta a saída da análise sintática executada pelo nosso analisador na entrada Teste54, Teste5 e Teste6:



```
krysthian@lessa: ~/MATA61-Compiladores/Analizador_Sintatico
Arquivo Editar Ver Pesquisar Terminal Ajuda
krysthian@lessa:~/MATA61-Compiladores/Analizador_Sintatico$ g++ sintatico.cpp
krysthian@lessa:~/MATA61-Compiladores/Analizador_Sintatico$ ./a.out < Arquivos_de_testes/Errados/teste04.txt
Analise léxica concluída sem erros!
Erro sintático em 14.12
Resolva o erro sintático para fazer a análise semântica.
krysthian@lessa:~/MATA61-Compiladores/Analizador_Sintatico$ ./a.out < Arquivos_de_testes/Errados/teste05.txt
Analise léxica concluída sem erros!
Erro sintático em 6.21
Resolva o erro sintático para fazer a análise semântica.
krysthian@lessa:~/MATA61-Compiladores/Analizador_Sintatico$ ./a.out < Arquivos_de_testes/Errados/teste06.txt
Analise léxica concluída sem erros!
Erro sintático em 8.18
Resolva o erro sintático para fazer a análise semântica.
krysthian@lessa:~/MATA61-Compiladores/Analizador_Sintatico$
```

Figure 9. Execução de analisador sintático

5. Análise Semântica

Na parte de análise semântica será verificado os erros semânticos gerados pelo programa [Berger] verificamos então se as declarações todas as variáveis usadas foram realmente declaradas? se todos os tipos da variável é o correto para o operador? se o fluxo de controle o comando é válido nesse contexto?

5.1. Regras Semânticas

Foram definidas algumas regras semânticas para a linguagem, que serão descritas aqui, são elas, as regras semânticas definidas para a linguagem não permite que: Existam duas declarações de instrução com o mesmo nome, e que Declarações **Vire Para** imediatamente subsequentes tenham sentidos diferentes, aqui temos alguns exemplos de como o compilador deve se comportar ao tratar os erros semânticos.

```
1 Exemplos:
2 Vire Para ESQUERDA (Permitido)
3 Vire Para ESQUERDA
4
5 Exemplo 2      Comando Mova: Esquerda e Esquerda.
6 Vire Para DIREITA (Permitido)
7 Vire Para DIREITA
8
9 Exemplo 3 - Comando Mova: Direita e Direita.
10 Vire Para ESQUERDA (Permitido)
11 Pare
12 Vire Para DIREITA
13 Exemplo 4      Comando Mova: Esquerda, Pare e Direita.
14 Vire Para ESQUERDA (N o Permitido)
15 Vire Para DIREITA
16
17 Exemplo 5      Comando Mova: Esquerda e Direita.
18 Vire Para DIREITA (N o Permitido)
19 Vire Para ESQUERDA
20
21 Exemplo 6      Comando Mova: Direita e Esquerda.
```

Como o robô é composto por dispositivos mecânicos, algumas instruções precisam ser concluídas para que assim novas instruções possam ser executadas, com isso logo após uma instrução **Mova n**, em que representa o número de passos, deve ser precedida por uma instrução do tipo **Aguarde até Pronto**;

5.1.1. Código fonte

Como foi feito anteriormente, vamos exibir o código gerado pelo analisador semântico e então logo em seguida detalhar os trechos de código gerado abaixo.

```
1 vector<Lexema> passarParaVector(list<Lexema> lexemas) {
2     vector<Lexema> vetor;
3
4     for (int i = 0; !lexemas.empty(); i++) {
5         vetor.push_back(lexemas.front());
6         lexemas.pop_front();
7     }
8
9     return vetor;
10 }
11
12 bool contemNaListaDeInstrucoes(string instrucao) {
13     for (int i = 0; i < instrucoesDeclaradas.size(); i++) {
14         if (instrucao == instrucoesDeclaradas[i])
15             return true;
16     }
```

```

17     return false;
18 }
19
20 bool analisadorSemantico(vector<Lexema> lexemas) {
21     bool flag = true;
22
23     for(int i = 0; i < lexemas.size(); i++) {
24         if (i > 0 && lexemas[i - 1].str == "definainstrucao") { //Declara o
25             de instru o
26             if (contemNaListaDeInstrucoes(lexemas[i].original)) {
27                 cout
28                 << "Erro Sem ntico " << lexemas[i].linha << "." << lexemas
29                 [i].coluna
30                 << ": Instru o j declarada." << endl;
31                 flag = false;
32             } else {
33                 instrucoesDeclaradas.push_back(lexemas[i].original);
34             }
35         }
36         if ((lexemas[i].str == "esquerda") && (lexemas[i - 1].str == "vire_para
37             ")) {
38             if ((lexemas[i - 3].str == "vire_para") && (lexemas[i - 2].str == "
39                 direita")) {
40                 cout
41                 << "Erro Sem ntico " << lexemas[i].linha << "." << lexemas
42                 [i].coluna
43                 << ": Declara es de Vire Para imediatamente sub. com
44                 sentidos diferentes." << endl;
45                 flag = false;
46             }
47         }
48         if ((lexemas[i].str == "direita") && (lexemas[i - 1].str == "vire_para"
49             )) {
50             if ((lexemas[i - 3].str == "vire_para") && (lexemas[i - 2].str == "
51                 esquerda")) {
52                 cout
53                 << "Erro Sem ntico " << lexemas[i].linha << "." << lexemas
54                 [i].coluna
55                 << ": Declara es de 'Vire Para' imediatamente
56                 subsequentes com sentidos diferentes." << endl;
57                 flag = false;
58             }
59         }
60         if ((lexemas[i].str == "passos") && (lexemas[i - 1].str == "num") &&
61             (lexemas[i - 2].str == "mova")) {
62             if (lexemas[i + 1].str != "aguarde_ate" && lexemas[i + 2].str != "
63                 robo_pronto") {
64                 cout
65                 << "Erro Sem ntico " << lexemas[i].linha << "." << lexemas
66                 [i].coluna
67                 << ": A instru o 'Mova n Passos' deve ser precedida por
68                 'Aguarde Ate Robo Pronto'." << endl;
69                 flag = false;
70             }
71         }
72     }
73     return flag;
74 }

```


Na análise semântica utilizamos uma estrutura de vetor de string para armazenar todas as instruções declaradas. Pois o algoritmo ao identificar uma declaração de instrução, ele verifica nessa estrutura se já não há uma instrução com o mesmo nome declarada, caso haja, o programa imprime o erro e continua a análise semântica. Para tratar os erros semânticos de instruções de "Vire Para" subsequentes com direções opostas, basicamente utilizamos desvios condicionais, com base em condições de tokens anteriores e posteriores. O mesmo foi feito para o tratamento de erros semânticos no caso da instrução "Mova n passos" que deveria ser precedida por um "Aguarde ate robo pronto" obrigatoriamente. Em algum desses três casos ocorrem, o programa imprime em qual linha e coluna ocorre.

5.2. Erros semânticos

Criamos também arquivos de entrada para testes do analisador semântico, para assim encontrar erros semânticos, abaixo temos 3 entradas (Teste1, Teste2, Teste3) que não possuem erros semânticos, portanto o analisador deve imprimir na tela a string "Análise Semântica concluída sem erros!". E 3 entradas (Teste3, Teste4, Teste5) que possuem erros léxicos, a qual o analisador semântico deve imprimir o erro semântico encontrado, bem como a linha e coluna onde o erro ocorre.

Primeiro vamos as entradas sem erros semanticos:

Teste1:

```
1 programainicio
2   definainstrucao compiladores como
3   inicio
4       vire para esquerda
5       acenda lampada
6       vire para direita
7   fim
8   definainstrucao compiladores2019 como
9   inicio
10      apague lampada
11      vire para esquerda
12  fim
13  execucaoinicio
14  inicio
15      repita 3 vezes compiladores fimrepita
16      vire para direita
17  fim
18  fimexecucao
19 fimprograma
```

Teste2:

```
1 programainicio
2   definainstrucao compiladores como
3   inicio
4       vire para esquerda
5       acenda lampada
6       vire para direita
7   fim
8   definainstrucao compiladores2019 como
9   inicio
10      vire para esquerda
11      se lampada acesa a direita entao
12          inicio
```

```

13         vire para esquerda
14         apague lampada
15     fim
16     fimse
17     vire para direita
18 fim
19 execucaoinicio
20 inicio
21     repita 3 vezes compiladores fimrepita
22     vire para direita
23     acenda lampada
24     vire para esquerda
25 fim
26 fimexecucao
27 fimprograma

```

Teste3:

```

1 programainicio
2     definainstrucao compiladores como
3     inicio
4         vire para esquerda
5         acenda lampada
6         vire para direita
7     fim
8     execucaoinicio
9     inicio
10        repita 3 vezes compiladores fimrepita
11        vire para direita
12        mova 5 passos
13        aguarde ate robo pronto
14        apague lampada
15    fim
16    fimexecucao
17 fimprograma

```

Agora os testes com erros semânticos:

Teste 4:

```

1     programainicio
2     definainstrucao compiladores como
3     inicio
4         vire para esquerda
5         acenda lampada
6         vire para direita
7     fim
8     definainstrucao compiladores como
9     inicio
10        apague lampada
11        vire para esquerda
12    fim
13    execucaoinicio
14    inicio
15        repita 3 vezes compiladores fimrepita
16        vire para direita
17    fim
18    fimexecucao
19 fimprograma

```

Teste 5:

```

1 programainicio
2   definainstrucao compiladores como
3   inicio
4     vire para esquerda
5     acenda lampada
6     vire para direita
7   fim
8   definainstrucao compiladores como
9   inicio
10    apague lampada
11    vire para esquerda
12    vire para direita
13  fim
14  execucaoinicio
15  inicio
16    repita 3 vezes compiladores fimrepita
17    vire para direita
18    vire para esquerda
19  fim
20  fimexecucao
21 fimprograma

```

Teste 6:

```

1 programainicio
2   definainstrucao compiladores como
3   inicio
4     vire para esquerda
5     acenda lampada
6     vire para direita
7   fim
8   definainstrucao compiladores como
9   inicio
10    apague lampada
11    vire para esquerda
12    vire para direita
13  fim
14  execucaoinicio
15  inicio
16    repita 3 vezes compiladores fimrepita
17    vire para direita
18    vire para esquerda
19    mova 5 passos
20    apague lampada
21  fim
22  fimexecucao
23 fimprograma

```

Na figura 8 abaixo temos o resultado da execução dos três primeiros testes acima, os códigos que estão sem erro semântico:

```
krysthian@lessa: ~/MATA61-Compiladores/Analizador_Semantico
Arquivo Editar Ver Pesquisar Terminal Ajuda
krysthian@lessa:~/MATA61-Compiladores/Analizador_Semantico$ g++ semantico.cpp
krysthian@lessa:~/MATA61-Compiladores/Analizador_Semantico$ ./a.out < Arquivos_de_testes/Corretos/teste01.txt
Análise léxica concluída sem erros!
Análise sintática concluída sem erros!
Análise semântica concluída sem erros!
krysthian@lessa:~/MATA61-Compiladores/Analizador_Semantico$ ./a.out < Arquivos_de_testes/Corretos/teste02.txt
Análise léxica concluída sem erros!
Análise sintática concluída sem erros!
Análise semântica concluída sem erros!
krysthian@lessa:~/MATA61-Compiladores/Analizador_Semantico$ ./a.out < Arquivos_de_testes/Corretos/teste03.txt
Análise léxica concluída sem erros!
Análise sintática concluída sem erros!
Análise semântica concluída sem erros!
krysthian@lessa:~/MATA61-Compiladores/Analizador_Semantico$
```

Figure 10. Execução Analizador Semântico, saídas corretas

Na figura 9 abaixo temos o resultado da execução dos outros 3 testes acima, que estão com erros semânticos:

```
krysthian@lessa: ~/MATA61-Compiladores/Analizador_Semantico
Arquivo Editar Ver Pesquisar Terminal Ajuda
krysthian@lessa:~/MATA61-Compiladores/Analizador_Semantico$ g++ semantico.cpp
krysthian@lessa:~/MATA61-Compiladores/Analizador_Semantico$ ./a.out < Arquivos_de_testes/Errados/teste04.txt
Análise léxica concluída sem erros!
Análise sintática concluída sem erros!
Erro Semântico 8.33: Instrução já declarada.

Resolva os erros semânticos para concluir compilação.
krysthian@lessa:~/MATA61-Compiladores/Analizador_Semantico$ ./a.out < Arquivos_de_testes/Errados/teste05.txt
Análise léxica concluída sem erros!
Análise sintática concluída sem erros!
Erro Semântico 8.33: Instrução já declarada.
Erro Semântico 12.26: Declarações de 'Vire Para' imediatamente subsequentes com sentidos diferentes.
Erro Semântico 18.27: Declarações de 'Vire Para' imediatamente subsequentes com sentidos diferentes.

Resolva os erros semânticos para concluir compilação.
krysthian@lessa:~/MATA61-Compiladores/Analizador_Semantico$ ./a.out < Arquivos_de_testes/Errados/teste06.txt
Análise léxica concluída sem erros!
Análise sintática concluída sem erros!
Erro Semântico 8.33: Instrução já declarada.
Erro Semântico 12.26: Declarações de 'Vire Para' imediatamente subsequentes com sentidos diferentes.
Erro Semântico 18.27: Declarações de 'Vire Para' imediatamente subsequentes com sentidos diferentes.
Erro Semântico 19.22: A instrução 'Mova n Passos' deve ser precedida por 'Aguarde Ate Robo Pronto'.

Resolva os erros semânticos para concluir compilação.
krysthian@lessa:~/MATA61-Compiladores/Analizador_Semantico$
```

Figure 11. Execução Analizador Semântico, saídas corretas

Como podemos ver o analisador semântico encontrou erro na linha 8 e coluna 33 teste 4, "Instrução já declarada.", ou seja, aquela instrução já foi dita anteriormente, no teste 5, além de ter erros de instrução já declarada tem-se também o erro de declaração, "Declarações de 'Vire Para' imediatamente subsequente com sentidos diferentes.", já no teste 6, temos os mesmos erros ditos anteriormente, e temos também o erro em que "A instrução 'Mova n Passos' deve ser precedida por 'Aguarde Ate Robo Pronto'".

6. Geração de código

A geração de código intermediário é basicamente a transformação da árvore de derivação em um segmento de código.[Ullman and Aho 1977] Esse código por sua vez pode eventualmente, ser o código objeto final, porém muitas vezes se constitui em um código intermediário. A diferença que existe entre o código intermediário e o código objeto final

é na especificação dos detalhes, o intermediário não especifica os detalhes da máquina alvo, como por exemplo quais registradores serão usados, ou quais endereços de memória serão referenciados.

6.1. Regras da Geração de código

Após terminado todas as outras fases o analisador léxico, sintático e semântico do código, o compilador então deve traduzir o código fonte para um código intermediário em linguagem de montagem, seguindo o padrão Intel 8086. O código assembly gerado pode ser executado usando o emulador de 8086, denominado emu8086. Nesse emulador, é considerada a possibilidade de comunicação com elementos externos através de portas de E/S (comandos e). O emulador suporta o interfaceamento com dispositivos virtuais que podem ser criados usando qualquer linguagem de programação que permita manipulação de arquivos.

O interfaceamento com tais dispositivos é feito usando um arquivo () para comunicação com o dispositivo virtual. A porta é representada pelo byte zero no arquivo, a porta pelo byte um, a porta pelo byte dois, e assim por diante. Através do arquivo podem-se endereçar portas de a (a).

No emu8086 existe a possibilidade de interfaceamento com um robô móvel (dispositivo virtual previamente disponível como exemplo no simulador). O robô é controlado pelo envio de dados para a porta de E/S de número 9. Considerando os comandos da Tabela 4.

Tabela 4 - Lista de comandos para o robô móvel

Valor_decimal	Valor_binario	Acao
0	00000000	N o executa qualquer acao.
1	00000001	Movimenta para frente.
2	00000010	Vira para esquerda.
3	00000011	Vira para direita.
4	00000100	Examina um objeto usando o sensor. Quando a tarefa finalizada, armazena o resultado no registrador de dados e um bit 1 no registrador de estado.
5	00000101	Acende uma lampada
6	00000110	Apaga uma lampada.
Um exemplo de transforma o pode ser vista no Exemplo 7.		

```
1 Mova MOV AL, 1 ;MOVA PARA FRENTE
2
3 OUT 9, AL
4
5 Vire Para Direita MOV AL, 3 ;VIRAR PARA DIREITA
6
7 OUT 9, AL
8
9 Mova MOV AL, 1 ;MOVA PARA FRENTE
10
11 OUT 9, AL
12
13 Vire Para Esquerda MOV AL, 2 ;VIRAR PARA ESQUERDA
14
15 OUT 9, AL
16
```

```

17 Mova MOV AL, 1 ;MOVA PARA FRENTE
18
19 OUT 9, AL
20
21 (a) Linguagem de Alto Nível (b) Linguagem de montagem
22 Exemplo 7 - Exemplo de transformação de código

```

As informações sobre a execução do comando “examinar” enviado para o robô pode ser obtido através do registrador de dados (porta 10). Como podemos ver abaixo:

Valor_decimal	Valor_binario	Significado
255	11111111	Parede a frente
0	00000000	Nada em frente
7	00000111	Lampada acessa a frente
8	00001000	Lampada apagada a frente
9	00001001	Lampada acessa a esquerda
10	00001010	Lampada apagada a esquerda
11	00001011	Lampada acessa a direita
12	00001100	Lampada apagada a direita
240	11110000	Parede a esquerda
15	00001111	Parede a direita

O registrador de estado atual do robô pode ser obtido usando o registrador de estado (porta 11), como podemos ver abaixo.

Numero_do_bit	Descricao
bit 0	Zero quando nao existem novos dados no registrador de dados e um caso contrario.
bit 1	Zero quando o robo esta pronto para receber novos comandos e um caso contrario.
bit 2	Zero quando nao existem nao ocorreram problemas na execucao do ultimo comando e um caso contrario.

7. Documentação e repositório

Toda documentação, exemplos, tabelas, autômatos e código fonte esta no seguinte repositório: github.com/krysthianlessa/MATA61-Compiladores

8. Referências

References

Berger, M. M. Compiladores.

Johnson, S. C. et al. (1975). *Yacc: Yet another compiler-compiler*, volume 32. Bell Laboratories Murray Hill, NJ.

Ullman, J. D. and Aho, A. V. (1977). Principles of compiler design. *Reading: Addison Wesley*.