Gustavo P. Gouveia (6482819), Victor Lassance (6431325)

Relatório de Compiladores

Terceira Etapa

Implementação do Reconhecedor Sintático Linguagem de programação CZAR

Texto apresentado à Escola Politécnica da Universidade de São Paulo como requisito para a aprovação na disciplina Linguagens e Compiladores no quinto módulo acadêmico do curso de graduação em Engenharia de Computação, junto ao Departamento de Engenharia de Computação e Sistemas Digitais (PCS).

Universidade de São Paulo

Escola Politécnica

Engenharia de Computação - Curso Cooperativo

Professor: Ricardo Luis de Azevedo da Rocha

São Paulo

2013

Resumo

Este trabalho descreve a concepção e o desenvolvimento de um compilador utilizando a linguagem C. O escopo do compilador se limita a casos mais simples, porém simbólicos, e que servem ao aprendizado do processo de criação e teste de um compilador completo. A estrutura da linguagem escolhida para ser implementada se assemelha a própria estrutura do C, por facilidade de compreensão, porém com algumas peculiaridades trazidas de outras linguagens.

Palavras-chaves: Linguagens, Compiladores, Implementação do Reconhecedor Sintático.

Sumário

1 Introdução

Este projeto tem como objetivo a construção de um compilador de um só passo, dirigido por sintaxe, com analisador e reconhecedor sintático baseado em autômato de pilha estruturado.

Em um primeiro momento, foi definida uma linguagem de programação e identificados os tipos de átomos. Para cada átomo foi escrito uma gramática linear representativa da sua lei de formação e um reconhecedor para o átomo. Desse modo, as gramáticas assim escritas foram unidas e convertidas em um autômato finito, o qual foi transformado em um transdutor e implementado como sub-rotina, dando origem ao analisador léxico propriamente dito. Também foi criada uma função principal para chamar o analisador léxico e possibilitar o seu teste.

Durante a segunda etapa, a sintaxe da linguagem, denonimada por nós de CZAR, foi definida formalmente a partir de uma definição informal e de exemplos de programas que criamos, misturando palavras-chave e conceitos de diferentes linguagens de programação. As três principais definições foram escritas na notação BNF[1], Wirth[2] e com diagramas de sintaxe.

Nessa etapa, implementamos o módulo referente à parte sintática para a nossa linguagem. O papel do analisador sintático é obter uma cadeia de *tokens* proveniente do analisador léxico, e verificar se a mesma pode ser gerada pela gramática da linguagem e, com isso, construir a árvore sintática (ALFRED; SETHI; JEFFREY, 1986).

Como material de consulta, além de sites sobre o assunto e das aulas ministradas, foi utilizado o livro indicado pelo professor no começo das aulas (NETO, 1987), para pesquisa de conceitos e possíveis implementações.

O documento apresenta a seguir as respostas às questões propostas para a terceira etapa.

[1] Ver http://en.wikipedia.org/wiki/Backus_Naur_Form
[2] Ver http://en.wikipedia.org/wiki/Wirth_syntax_notation

2 Descrição Atualizada da Linguagem em Wirth

```
1  PROGRAM          = IMPORTS DECLS_GLOBAIS DEF_PROCS_FUNCS "main" DEF_MAIN.
2  IMPORTS          = { "<" IDENT ">" } .
3
4  DECLS_GLOBAIS    = { "struct" IDENT "{" { [ "const" ] IDENT { "[" INT "]" } IDENT
       [ "=" EXPR ] { "," IDENT [ "=" EXPR ] } ";" } "}" | [ "const" ] IDENT { "["
       INT "]" } IDENT [ "=" EXPR ] { "," IDENT [ "=" EXPR ] } ";" } "meth".
5
6  DEF_PROCS_FUNCS = { "void" IDENT LIST_PARAMS "{" { INSTR_SEM_RET } "}" | IDENT {
       "[" INT "]" } IDENT LIST_PARAMS "{" { INSTR_SEM_RET } "return" EXPR [ ";" ]
       "}" }.
7  LIST_PARAMS      = "(" [ [ "ref" ] IDENT { "[" INT "]" } IDENT { "," [ "ref" ]
      IDENT { "[" INT "]" } IDENT } ] ")".
8
9  DEF_MAIN         = "(" ")" "{" { INSTR_SEM_RET } "}".
10
11 INSTR_SEM_RET    = IDENT ( "[" ( [ "+" | "-" ] ( "(" EXPR ")" |( INT | FLOAT |
      IDENT ( "(" [ EXPR { "," EXPR } ] ")" | { "[" EXPR "]" } { "." VARIDENT } ))
       ) | STR | CHAR | BOOL { "*" | "/" | "%" ATOMO } { ( "+" | "-" ) TERM} "]" {
       "[" EXPR "]" } { "." VARIDENT } ["+" | "-" | "*" | "/" | "%" ] "=" EXPR {
      "," VARIDENT OPER_ATRIB EXPR } ";" | INT "]" { "[" INT "]" } IDENT [ "="
      EXPR ] { "," IDENT [ "=" EXPR ] } ";" ) | IDENT [ "=" EXPR ] { "," IDENT [
      "=" EXPR ] } ";" | { "." VARIDENT } [ "+" | "-" | "*" | "/" | "%"] "=" EXPR
      { "," VARIDENT OPER_ATRIB EXPR } ";" | "(" [ EXPR { "," EXPR } ] ")" ";" ) |
       "for" "(" IDENT { "[" INT "]" } IDENT [ "=" EXPR ] { "," IDENT [ "=" EXPR ]
       } ";" COND ";" VARIDENT OPER_ATRIB EXPR { "," VARIDENT OPER_ATRIB EXPR } ")
      " "{" { INSTR_SEM_RET } "}" | "while" "(" COND ")" "{" { INSTR_SEM_RET } "}"
       | "if" "(" COND ")" "{" { INSTR_SEM_RET } "}" ["else" "{" { INSTR_SEM_RET }
       "}"].
12
13 VARIDENT         = IDENT { "[" EXPR "]" } { "." IDENT { "[" EXPR "]" } } .
14
15 FUNCTION_CALL    = IDENT "(" [ EXPR { "," EXPR } ] ")".
16
17 BOTH = IDENT ( { "[" EXPR "]" } { "." IDENT { "[" EXPR "]" } } | "(" [ EXPR {
      "," EXPR } ] ")").
18
19 COND             = COND_TERM { ("and" | "or") COND_TERM}.
20 COND_TERM        = "(" COND ")" | ATOMO_COND { ("==" | "!=" | "<=" | ">=")
      ATOMO_COND}.
21 ATOMO_COND       = VARIDENT | BOOL | INT | "not" ATOMO_COND.
22
23 OPER_ATRIB       =  "+=" | "-=" | "*=" | "/=" | "%=" | "=".
24
25 EXPR             = [ "+" | "-" ] TERM { ( "+" | "-" ) TERM } .
26 TERM             = "(" EXPR ")" | ATOMO { ( "*" | "/" | "%" ) ATOMO } .
27 ATOMO            = ([ "+" | "-" ] ( BOTH | INT | FLOAT )) | STR | CHAR | BOOL .
```

3 Lista de Submáquinas do APE

3.1 Lista de Transições

Autômato Finito Determinístico Mínimo da submáquina ATOMO-COND:

```
initial: 0
final: 1
(0, VARIDENT) -> 1
(0, BOOL) -> 1
(0, INT) -> 1
(0, "not") -> 2
(2, ATOMO_COND) -> 1
```

Autômato Finito Determinístico Mínimo da submáquina *ATOMO*:

```
initial: 0
final: 2
(0, "+") -> 1
(0, "-") -> 1
(0, BOTH) -> 2
(0, INT) -> 2
(0, FLOAT) -> 2
(0, STR) -> 2
(0, CHAR) -> 2
(0, BOOL) -> 2
(1, BOTH) -> 2
(1, INT) -> 2
(1, FLOAT) -> 2
```

Autômato Finito Determinístico Mínimo da submáquina *BOTH*:

```
initial: 0
final: 1, 7, 8
(0, IDENT) -> 1
(1, "[") -> 2
(1, ".") -> 3
(1, "(") -> 4
(2, EXPR) -> 5
(3, IDENT) -> 8
(4, EXPR) -> 6
(4, ")") -> 7
(5, "]") -> 8
(6, ",") -> 9
(6, ")") -> 7
(8, "[") -> 2
(8, ".") -> 3
(9, EXPR) -> 6
```

Autômato Finito Determinístico Mínimo da submáquina *COND-TERM*:

```
initial: 0
```

```
final: 2, 5
(0, "(") -> 1
(0, ATOMO_COND) -> 2
(1, COND) -> 3
(2, "==") -> 4
(2, "!=") -> 4
(2, "<=") -> 4
(2, ">=") -> 4
(3, ")") -> 5
(4, ATOMO_COND) -> 2
```

Autômato Finito Determinístico Mínimo da submáquina *COND*:

```
initial: 0
final: 1
(0, COND_TERM) -> 1
(1, "and") -> 0
(1, "or") -> 0
```

Autômato Finito Determinístico Mínimo da submáquina *DECLS-GLOBAIS*:

```
initial: 0
final: 4
(0, "struct") -> 1
(0, IDENT) -> 2
(0, "const") -> 3
(0, "meth") -> 4
(1, IDENT) -> 5
(2, IDENT) -> 9
(2, "[") -> 10
(3, IDENT) -> 2
(5, "{") -> 6
(6, IDENT) -> 7
(6, "const") -> 8
(6, "}") -> 0
(7, IDENT) -> 12
(7, "[") -> 13
(8, IDENT) -> 7
(9, "=") -> 15
(9, ",") -> 16
(9, ";") -> 0
(10, INT) -> 11
(11, "]") -> 2
(12, "=") -> 17
(12, ",") -> 18
(12, ";") -> 6
(13, INT) -> 14
(14, "]") -> 7
(15, EXPR) -> 19
(16, IDENT) -> 9
(17, EXPR) -> 20
(18, IDENT) -> 12
(19, ",") -> 16
(19, ";") -> 0
(20, ",") -> 18
(20, ";") -> 6
```

Autômato Finito Determinístico Mínimo da submáquina *DEF-MAIN*:

```
initial: 0
final: 4
(0, "(") -> 1
(1, ")") -> 2
(2, "{") -> 3
(3, INSTR_SEM_RET) -> 3
(3, "}") -> 4
```

Autômato Finito Determinístico Mínimo da submáquina *DEF-PROCS-FUNCS*:

```
initial: 0
final: 9
(0, "void") -> 1
(0, IDENT) -> 2
(1, IDENT) -> 3
(2, IDENT) -> 4
(2, "[") -> 5
(3, LIST_PARAMS) -> 6
(4, LIST_PARAMS) -> 10
(5, INT) -> 7
(6, "{") -> 8
(7, "]") -> 2
(8, INSTR_SEM_RET) -> 8
(8, "}") -> 9
(9, "void") -> 1
(9, IDENT) -> 2
(10, "{") -> 11
(11, INSTR_SEM_RET) -> 11
(11, "return") -> 12
(12, EXPR) -> 13
(13, "}") -> 9
(13, ";") -> 14
(14, "}") -> 9
```

Autômato Finito Determinístico Mínimo da submáquina *EXPR*:

```
initial: 0
final: 2
(0, "+") -> 1
(0, "-") -> 1
(0, TERM) -> 2
(1, TERM) -> 2
(2, "+") -> 1
(2, "-") -> 1
```

Autômato Finito Determinístico Mínimo da submáquina *FUNCTION-CALL*:

```
initial: 0
final: 4
(0, IDENT) -> 1
(1, "(") -> 2
(2, EXPR) -> 3
```

```
(2, ")") -> 4
(3, ",") -> 5
(3, ")") -> 4
(5, EXPR) -> 3
```

Autômato Finito Determinístico Mínimo da submáquina *IMPORTS*:

```
initial: 0
final: 3
(0, "<") -> 1
(1, IDENT) -> 2
(2, ">") -> 3
(3, "<") -> 1
```

Autômato Finito Determinístico Mínimo da submáquina *INSTR-SEM-RET*:

```
initial: 0
final: 15, 19, 32, 35, 62, 66
(0, IDENT) -> 1
(0, "for") -> 2
(0, "while") -> 3
(0, "if") -> 4
(1, IDENT) -> 5
(1, "[") -> 6
(1, "+") -> 7
(1, "-") -> 7
(1, "(") -> 8
(1, ".") -> 9
(1, "*") -> 7
(1, "/") -> 7
(1, "%") -> 7
(1, "=") -> 10
(2, "(") -> 28
(3, "(") -> 23
(4, "(") -> 11
(5, ",") -> 53
(5, "=") -> 54
(5, ";") -> 15
(6, IDENT) -> 32
(6, "+") -> 33
(6, "-") -> 33
(6, "(") -> 34
(6, INT) -> 35
(6, FLOAT) -> 15
(6, STR) -> 15
(6, CHAR) -> 15
(6, BOOL) -> 36
(7, "=") -> 10
(8, EXPR) -> 25
(8, ")") -> 26
(9, VARIDENT) -> 22
(10, EXPR) -> 12
(11, COND) -> 13
(12, ",") -> 14
(12, ";") -> 15
```

```
(13, ")") -> 16
(14, VARIDENT) -> 18
(16, "{") -> 17
(17, INSTR_SEM_RET) -> 17
(17, "}") -> 19
(18, OPER_ATRIB) -> 10
(19, "else") -> 20
(20, "{") -> 21
(21, INSTR_SEM_RET) -> 21
(21, "}") -> 15
(22, "+") -> 7
(22, "-") -> 7
(22, ".") -> 9
(22, "*") -> 7
(22, "/") -> 7
(22, "%") -> 7
(22, "=") -> 10
(23, COND) -> 24
(24, ")") -> 20
(25, ")") -> 26
(25, ",") -> 27
(26, ";") -> 15
(27, EXPR) -> 25
(28, IDENT) -> 29
(29, IDENT) -> 30
(29, "[") -> 31
(30, ",") -> 41
(30, "=") -> 42
(30, ";") -> 43
(31, INT) -> 37
(32, "[") -> 59
(32, "(") -> 60
(32, ".") -> 61
(33, IDENT) -> 32
(33, "(") -> 34
(33, INT) -> 15
(33, FLOAT) -> 15
(34, EXPR) -> 67
(35, "]") -> 56
(36, "+") -> 38
(36, "-") -> 38
(36, "]") -> 39
(36, "*") -> 36
(36, "/") -> 36
(36, "%") -> 40
(37, "]") -> 29
(38, TERM) -> 52
(39, "[") -> 46
(39, "+") -> 7
(39, "-") -> 7
(39, ".") -> 9
(39, "*") -> 7
(39, "/") -> 7
(39, "%") -> 7
(39, "=") -> 10
(40, ATOMO) -> 36
(41, IDENT) -> 30
```

```
(42, EXPR) -> 50
(43, COND) -> 44
(44, ";") -> 45
(45, VARIDENT) -> 47
(46, EXPR) -> 51
(47, OPER_ATRIB) -> 48
(48, EXPR) -> 49
(49, ")") -> 20
(49, ",") -> 45
(50, ",") -> 41
(50, ";") -> 43
(51, "]") -> 39
(52, "+") -> 38
(52, "-") -> 38
(52, "]") -> 39
(53, IDENT) -> 5
(54, EXPR) -> 55
(55, ",") -> 53
(55, ";") -> 15
(56, IDENT) -> 5
(56, "[") -> 57
(57, INT) -> 58
(58, "]") -> 56
(59, EXPR) -> 65
(60, EXPR) -> 63
(60, ")") -> 15
(61, VARIDENT) -> 62
(62, ".") -> 61
(63, ")") -> 15
(63, ",") -> 64
(64, EXPR) -> 63
(65, "]") -> 66
(66, "[") -> 59
(66, ".") -> 61
(67, ")") -> 15
```

Autômato Finito Determinístico Mínimo da submáquina *LIST-PARAMS*:

```
initial: 0
final: 4
(0, "(") -> 1
(1, "ref") -> 2
(1, IDENT) -> 3
(1, ")") -> 4
(2, IDENT) -> 3
(3, IDENT) -> 5
(3, "[") -> 6
(5, ",") -> 7
(5, ")") -> 4
(6, INT) -> 8
(7, "ref") -> 2
(7, IDENT) -> 3
(8, "]") -> 3
```

Autômato Finito Determinístico Mínimo da submáquina *OPER-ATRIB*:

```
initial: 0
final: 1
(0, "+=") -> 1
(0, "-=") -> 1
(0, "*=") -> 1
(0, "/=") -> 1
(0, "%=") -> 1
(0, "=") -> 1
```

Autômato Finito Determinístico Mínimo da submáquina *PROGRAM*:

```
initial: 0
final: 5
(0, IMPORTS) -> 1
(1, DECLS_GLOBAIS) -> 2
(2, DEF_PROCS_FUNCS) -> 3
(3, "main") -> 4
(4, DEF_MAIN) -> 5
```

Autômato Finito Determinístico Mínimo da submáquina *TERM*:

```
initial: 0
final: 2, 5
(0, "(") -> 1
(0, ATOMO) -> 2
(1, EXPR) -> 3
(2, "*") -> 4
(2, "/") -> 4
(2, "%") -> 4
(3, ")") -> 5
(4, ATOMO) -> 2
```

Autômato Finito Determinístico Mínimo da submáquina *VARIDENT*:

```
initial: 0
final: 1
(0, IDENT) -> 1
(1, "[") -> 2
(1, ".") -> 0
(2, EXPR) -> 3
(3, "]") -> 1
```

3.2 Lista de Autômatos

- ATOMO-COND:

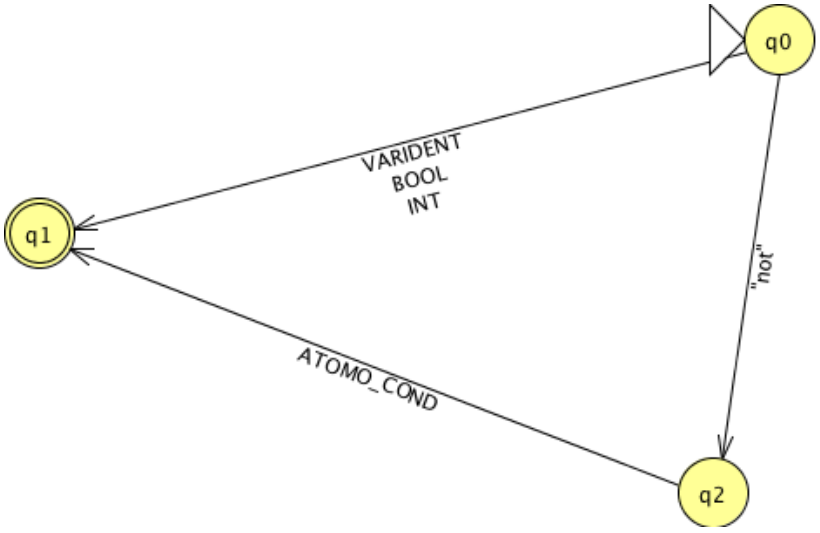

Figura 1 – Autômato ATOMO-COND

- ATOMO:

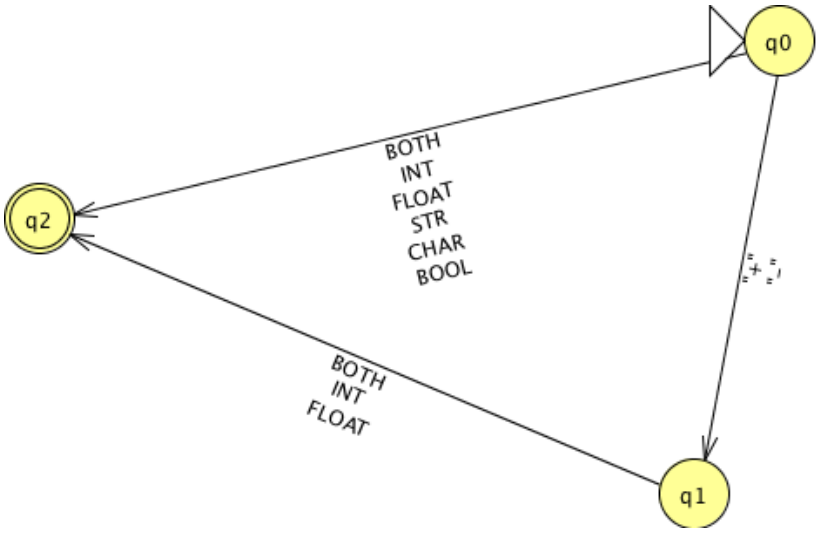

Figura 2 – Autômato ATOMO

- BOTH:

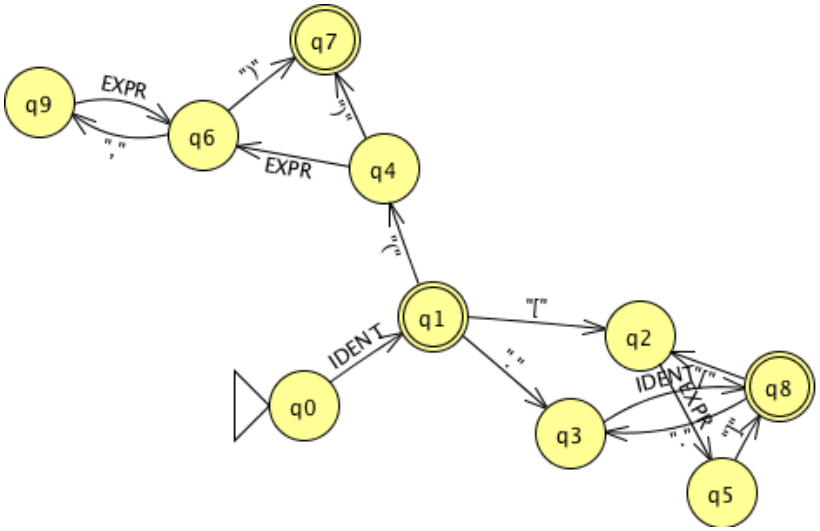

Figura 3 – Autômato BOTH

- COND-TERM:

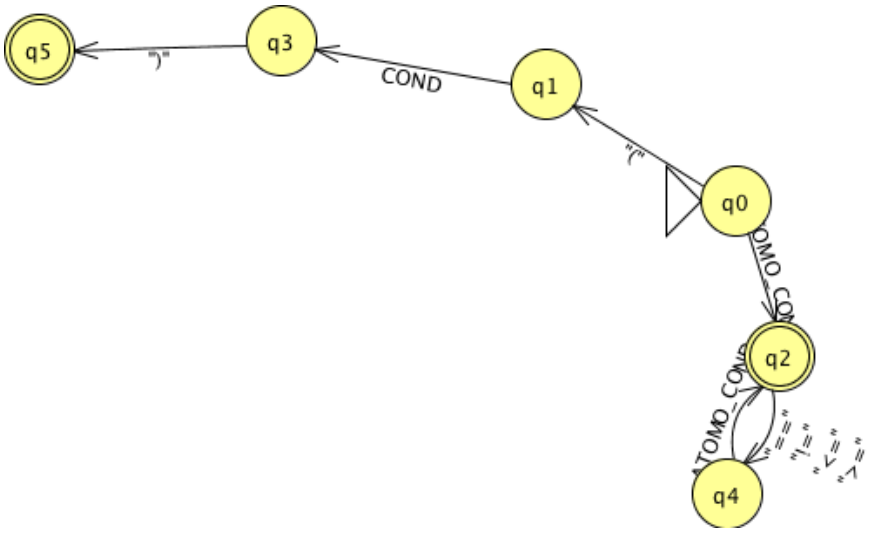

Figura 4 – Autômato COND-TERM

- COND:

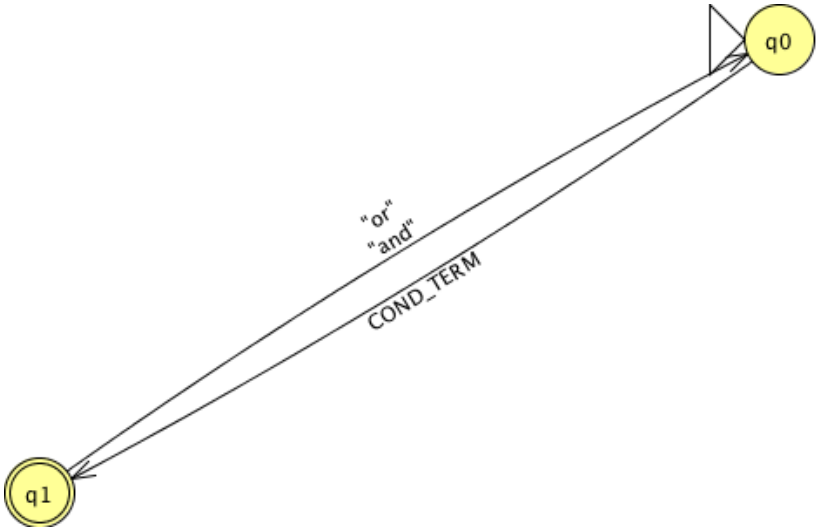

Figura 5 – Autômato COND

- DECLS-GLOBAIS:

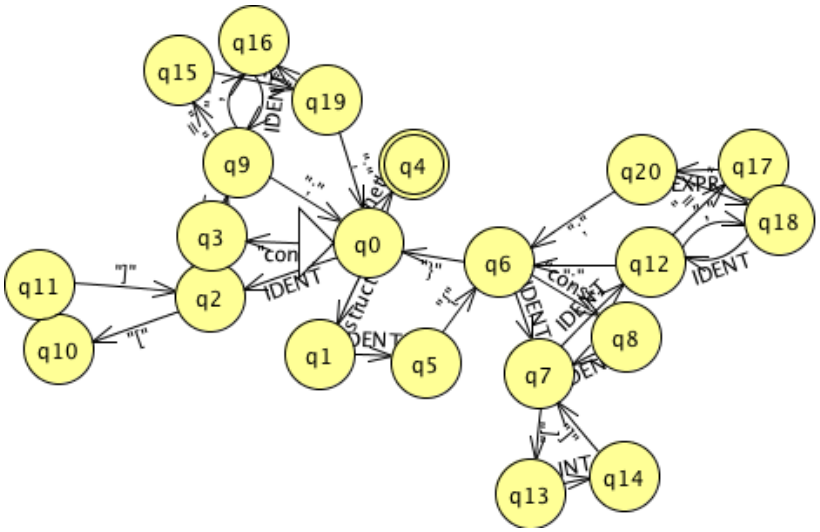

Figura 6 – Autômato DECLS-GLOBAIS

- DEF-MAIN:

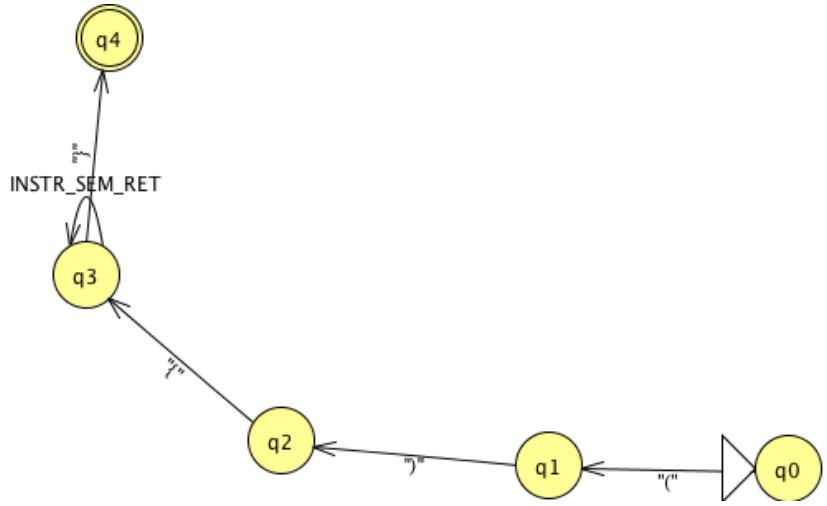

Figura 7 – Autômato DEF-MAIN

- DEF-PROCS-FUNCS:

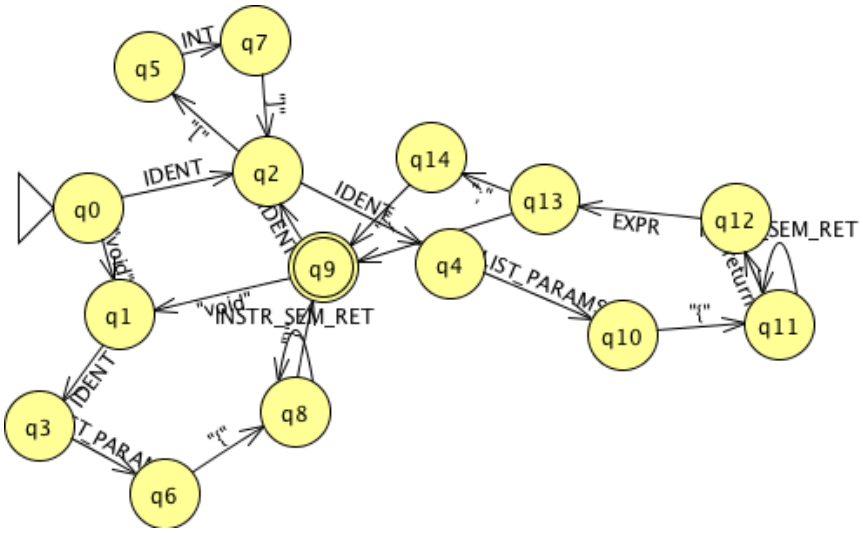

Figura 8 – Autômato DEF-PROCS-FUNCS

- EXPR:

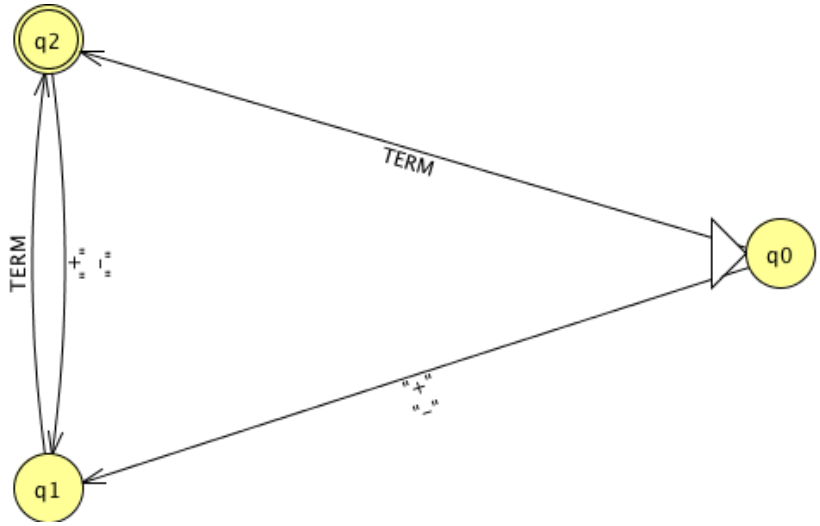

Figura 9 – Autômato EXPR

- FUNCTION-CALL:

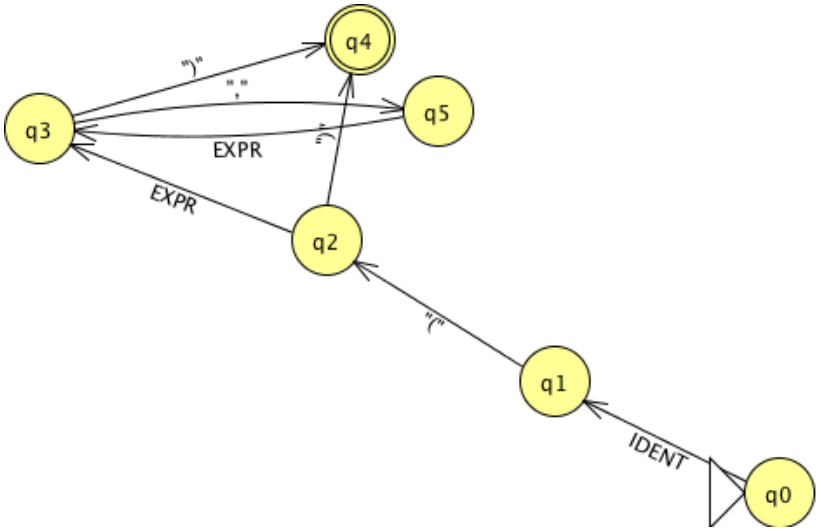

Figura 10 – Autômato FUNCTION-CALL

- IMPORTS:

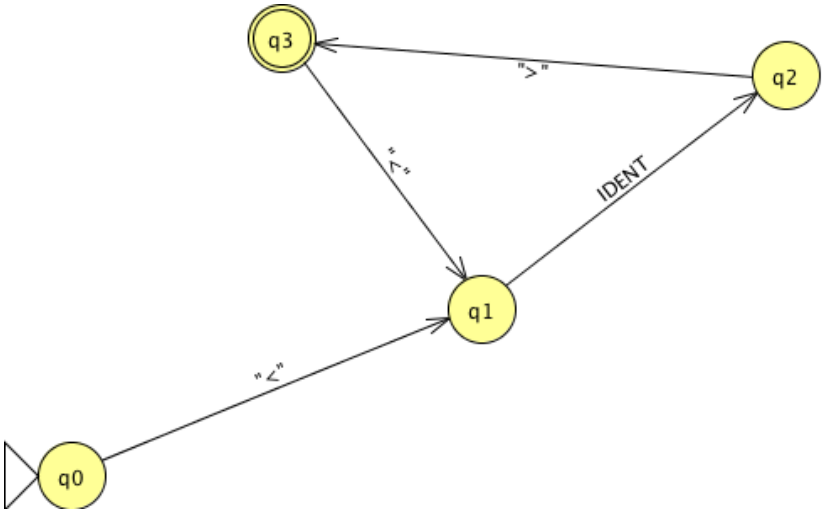

Figura 11 – Autômato IMPORTS

- INSTR-SEM-RET:

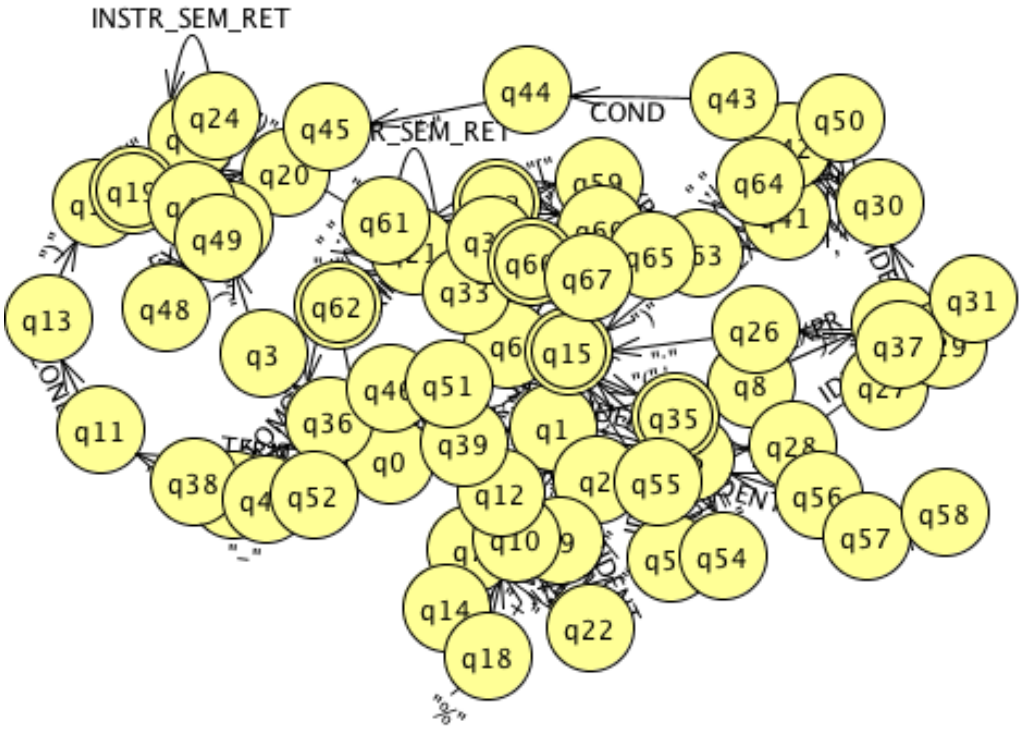

Figura 12 – Autômato INSTR-SEM-RET

- LIST-PARAMS:

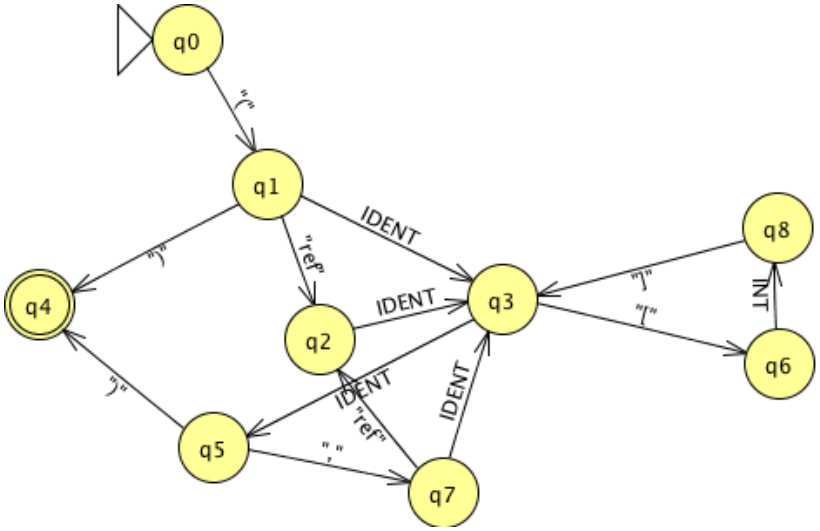

Figura 13 – Autômato LIST-PARAMS

- OPER-ATRIB:

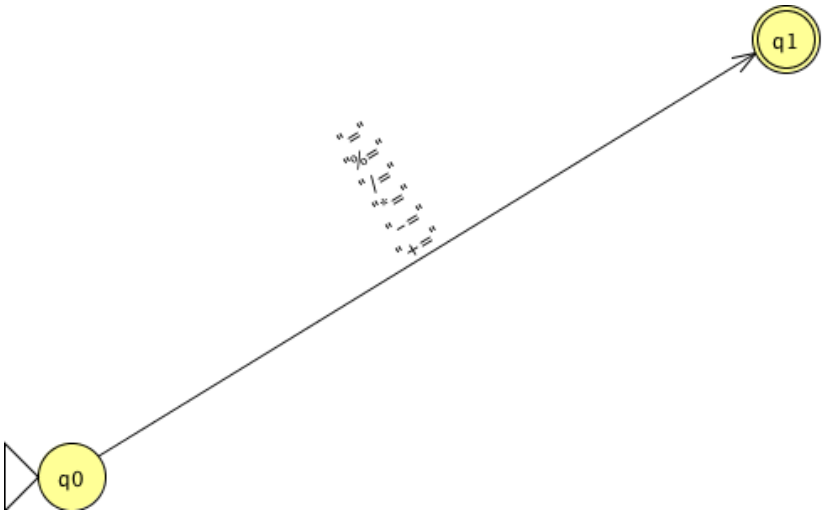

Figura 14 – Autômato OPER-ATRIB

- PROGRAM:

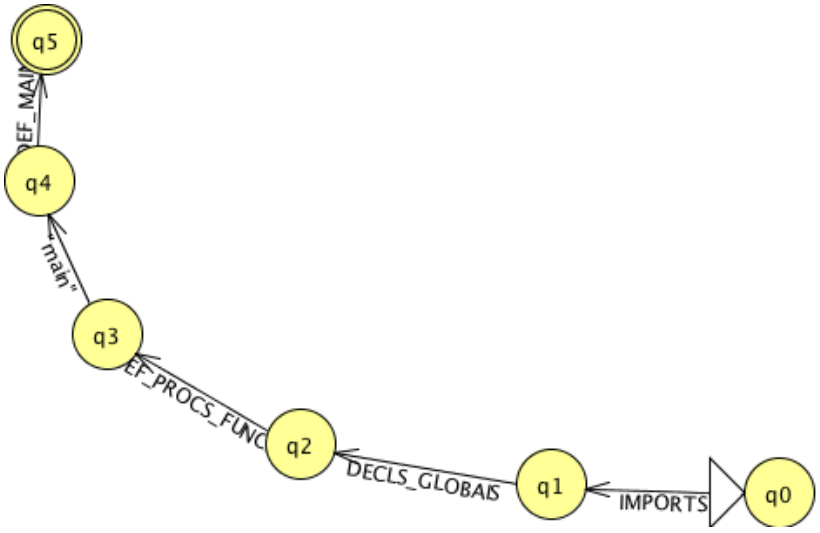

Figura 15 – Autômato PROGRAM

- TERM:

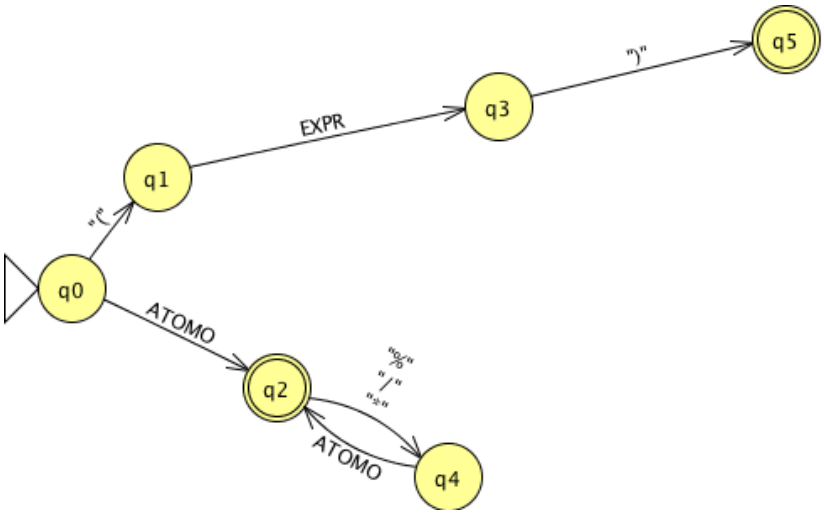

Figura 16 – Autômato TERM

- VARIDENT:

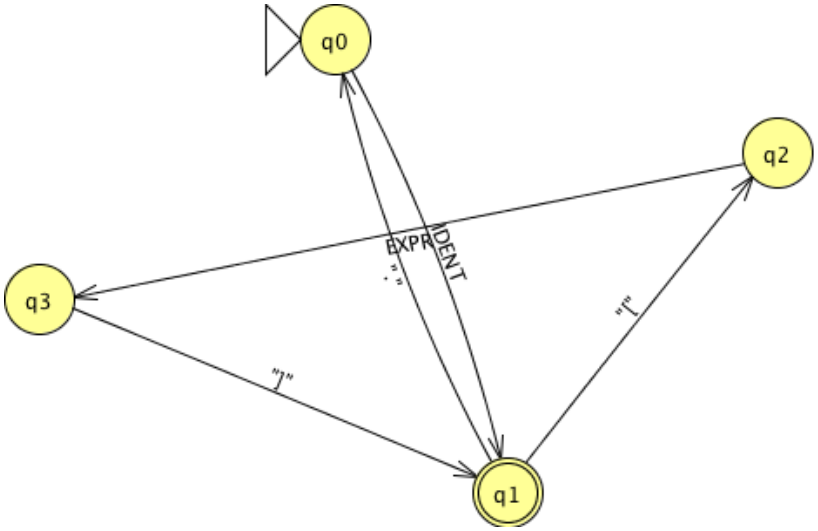

Figura 17 – Autômato VARIDENT

4 Exemplos Atualizados de Programas na Linguagem *CZAR*

4.1 Exemplo Geral

```
1  <math>
2  <io>
3
4  struct nome_struct {
5    nome_struct eu_mesmo;
6    int a;
7    char b;
8  }
9
10 const int SOU_CONSTANTE_INT = 10;
11 const string SOU_CONSTANTE_STRING = "CONSTANTE_STRING";
12 string sou_variavel = "valor inicial da variavel";
13
14 meth
15
16 void soma_como_procedimento (int a, int b, ref int soma) {
17   soma = a + b;
18 }
19
20 int soma_como_funcao (int a, int b) {
21   return a + b;
22 }
23
24 void proc_exemplo (char a, int b, int c, int d) {
25   int tmp;
26   char[32] buff;
27   soma_como_procedimento(b, c, tmp);
28   d = soma_como_funcao(tmp, c) + 5;
29   d = math_exp(SOU_CONSTANTE_INT, 2);
30   io_print(a);
31   io_int_to_str(d, buff);
32   io_print(" gives ");
33   io_print(buff);
34   io_print(" \n pointer to a is: ");
35   buff = a + "caracter";
36   io_print(buff);
37   io_print("bye");
38 }
39
40 main () {
41   proc_exemplo('x', 3, -6, -15);
42 }
```

4.2 Exemplo Fatorial

```
1  <io>
2
3  const int fat_10_rec = 10;
4  const int fat_10_iter = 10;
5  int retorno;
6
7  meth
8
9  int fatorial_recursivo(int n) {
10     int retorno = 1;
11         if (n >= 1) {
12         retorno = n * fatorial_recursivo (n - 1);
13     }
14     return retorno;
15 }
16
17 int fatorial_iterativo(int n) {
18     int fatorial = 1;
19     while (n >= 0) {
20         fatorial = fatorial * n;
21         n = n - 1;
22     }
23     return fatorial;
24 }
25
26 main () {
27     retorno = fatorial_recursivo(fat_10_rec);
28
29     io_print_int(retorno);
30     io_print(" ");
31     io_print_int(fatorial_iterativo(fat_10_iter));
32 }
```

5 Comentários sobre a Implementação do Reconhecedor Sintático

O analisador sintático foi escrito baseado nas máquinas de estado reduzidas geradas com o auxílio do site wirth.heroku.com, hospedando o mesmo aplicativo indicado pelo exercício.

A estratégia utilizada foi a construção de uma estrutura de dados de autômato, com 3 tipos de transições:

1. Final: Acrescidos de um espaço em sua identificação na máquina de estados (como se fossem submáquinas), estes são as classes definidas pelo léxico, como `STR` ou `NUMERO`
2. Transições: São os termos finais da notação de WIRTH que podem ser especificados, tais quais sinais de pontuação (`{}[]()`), palavras reservadas (main, void, ...) e outros.
3. Chamadas: São nomes associados às outras máquinas de estados. Estas máquinas podem ser chamadas, ocasionando o empilhamento da máquina atual em seu estado de retorno.

Como a indexação das máquinas de estados e dos próprios estados é baseada em um inteiro de 32 bits (`u_int32`), podemos construir uma pilha com um array simples de inteiros de 64 bits, fazendo deslocamentos para inserção dos valores.

5.0.1 Principais chamadas

- syn

  ```
  uint32_t syn(Token* tk, Automaton** a, uint32_t* state)
  ```

  Resulta na leitura ou não (retorno 1 ou 0 da função) do token `tk` e associação do autômato em `a`, assim como de seu estado em `state`.

- read_mdfa

  ```
  void read_mdfa(Automaton* a, char* name, uint32_t id, FILE* f)
  ```

Causa a leitura de um arquivo `*.mdfa` designado por `f` e a construção do autômato em `a`, alocando toda a memória necessária, assim como gravando o nome e identificador numérico do mesmo.

Referências

ALFRED, V.; SETHI, R.; JEFFREY, D. *Compilers: principles, techniques and tools.* [S.l.]: Addison-Wesley, 1986.

NETO, J. J. *Introdução à Compilação.* [S.l.]: LTC, 1987. (ENGENHARIA DE COMPUTAÇÃO).