Victor Lassance (6431325)

Relatório de Compiladores
Segunda Prova
Compilador de *SimpPro* para *RNA*

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

Sumário

Sumário **2**

1 Apresentação da linguagem *SimpPro* e enunciado **3**

2 Apresentação da linguagem *RNA* **5**

3 Analisador léxico **7**

4 Analisador sintático **9**

5 Analisador semântico **11**

6 Ambiente de execução e geração de código **14**

7 Testes e exemplos de execução **16**

7.1 Programa 1 16

7.2 Programa 2 16

1 Apresentação da linguagem *SimpPro* e enunciado

A linguagem *SimpPro* foi criada e apresentada pelo professor da disciplina com características similares a de outras linguagens. As principais linguagens herdadas pela *SimpPro* foram de Prolog, na forma de declaração e busca; e Lisp, na utilização dos parêntesis para declarar predicados, cláusulas e a meta.

A linguagem Prolog é declarativa, o seu texto pode conter variáveis (identificáveis lexicamente) ou nomes e números (constantes) ao estilo LISP. O operador de definição de termos é ":-", para uma verificação de meta o operador é "?-". Um programa em Prolog é composto usualmente de três partes: conjuntos de fatos, conjuntos de cláusulas e conjuntos de metas. Os fatos são dados sobre os quais é possível efetuar uma busca por meio de unificação de literais. As cláusulas representam a forma como os elementos de dados são inter-relacionados, definem predicados, seu uso por outros predicados e a relação entre predicados e fatos. As metas definem que tipo de resultado é esperado, podendo ser um resultado booleano, um conjunto de valores possíveis para uma variável, etc.

Para este exercício não será utilizada a linguagem Prolog completa, apenas um subconjunto bastante limitado e simplificado denominado *SimpPro*.

A sintaxe de SimpPro fornecida em BNF foi a seguinte:

```
<PROGRAMA> ::= <FATOS> <CLUSULAS> <METAS>
<FATOS> ::= ( <FATO> ) <FATOS> | ( <FATO> )
<CLUSULAS> ::= ( <CLUSULA> ) <CLUSULAS> | ( <CLUSULA> )
<METAS> ::= ( ?- <PRED> <DADO> )
<FATO> ::= <NOME> :- <DADO>
<DADO> ::= <NOME> , <DADO> | <NUM> , <DADO> | <NOME> | <NUM>
<CLUSULA> ::= <PRED> <ARGS> :- ( <LCLUSULA> ) | <PRED> <ARGS> :- <DADO>

<LCLUSULA> ::= <OP U> ( <INF> <DADO> ) <OP B> <LCLUSULA> | <OP U> ( <INF> <DADO> )
<ARGS> ::= <INF> , <ARGS> | <NOME> , <ARGS> | <NUM> , <ARGS> | <INF>
<PRED> ::= <NOME>

<INF> ::= <NOME> // INF inicia por letra maiuscula
<NOME> ::= <LETRA> | <DIGITO> <NOME> | <LETRA> <NOME>
<NUM> ::= <DIGITO> | <DIGITO> <NUM>
<OP B> ::= & | or
<OP U> ::= not | eps
<LETRA> ::= A | B | ... | Z | a | b | ... | z
<DIGITO> ::= 0 | 1 | ... | 9
```

Considerando que a unificação é feita através de uma busca em base dados (cuja implementação é conhecida e acessível) e que haverá apenas uma meta por programa, cujo resultado será booleano, ou seja, cada programa retornará verdadeiro (1) ou falso (0)

para a meta (que será uma cláusula completa), pede-se para construir um reconhecedor determinístico, baseado no autômato de pilha estruturado, que aceite como entrada válida um programa escrito em *SimpPro*.

Além disso, deve-se construir o sistema de programação para a linguagem *SimpPro*, que terá um compilador para a linguagem *RNA* com um ambiente de execução e uma função de busca para a meta definida. Deve ser usado a implementação de *RNA* feita em linguagem C para validar o código gerado pelo compilador, aceitando ou não a meta como inferência lógica dos fatos e das cláusulas.

2 Apresentação da linguagem *RNA*

A linguagem de programação *RNA* apresentada pelo professor é uma linguagem esotérica e nunca utilizada para aplicações práticas, criada em 2008 e implementada em 2011 por Cyrus H.

Ela possui 16 instruções implementadas e 3 variáveis para armazenamento de memória e processamento de dados, *strg*, *ptr* e *memory*. Como a *strg*, variável responsável por guardar o índice para acesso ao *memory* tem 8 bits, só podemos acessar 256 células de 8 bits cada uma, tendo uma memória bem limitada.

A Figura 1 mostra a relação das 3 variáveis.

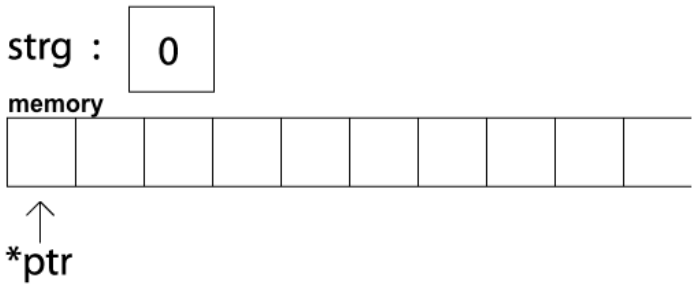

Figura 1 – Ilustração das estruturas de armazenamento do *RNA*

As 16 instruções implementadas correspondem às seguintes instruções em C:

1. **AUG**: `int main() {`
2. **UAA**: `} // end_main`
3. **UGG**: `strg=0;`
4. **AAA**: `++strg;`
5. **AAC**: `--strg;`
6. **GCA**: `strg=*ptr;`
7. **ACA**: `ptr=&memory[strg];`
8. **CCA**: `scanf("%d", ptr);`
9. **CUA**: `printf("%c", *ptr);`
10. **AGA**: `*ptr+=memory[strg];`

11. **AGC**: `*ptr*=memory[strg];`
12. **CAA**: `*ptr-=memory[strg];`
13. **CAC**: `*ptr/=memory[strg];`
14. **GAA**: `*ptr=*ptr==memory[strg]?1:0;`
15. **GAC**: `while(*ptr) {`
16. **UAC**: `} // end_while`

Com relação a implementação em C da linguagem[1], foram encontrados alguns erros que foram corrigidos a fim de permitir o teste de programas em RNA. Abaixo, segue o *diff* do que foi modificado com relação à implementação original.

```
326c326
<                 loopy=loopActivators[1][loopActivatorIndex--];
---
>                 loopy=loopActivators[1][loopActivatorIndex++]-1;
331c331
<                 loopActivators[0][loopActivatorIndex]=loopy-2;
---
>                 loopActivators[0][loopActivatorIndex++]=loopy-2;
337,338c337,338
<                 loopActivators[1][loopActivatorIndex]=codon+1;
<                 loopy=loopActivators[0][loopActivatorIndex--];
---
>                 loopActivators[1][--loopActivatorIndex]=loopy+1;
>                 loopy=loopActivators[0][loopActivatorIndex]-1;
```

A implementação do interpretador *RNA* com as correções pode ser encontrada junto com o código final, para permitir a realização de testes.

[1] http://esolangs.org/wiki/RNA

3 Analisador léxico

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. Cabe ressaltar que foi utilizado o analisador léxico do trabalho como base para esse, visto que a estrutura e alguns *tokens* eram os mesmos. Algumas das alterações feitas para adaptar o transdutor foram:

- **IDENT vs PRED + INF**: Antes, só havíamos um *token* para representar identificadores de alguma forma, chamados de **IDENT**. Porém, ao observar a sintaxe do *SimpPro*, foi necessário alterar o léxico para diferenciar tokens que começam com minúscula (chamado **PRED**) ou maiúscula (chamado **INF**);
- Os operadores específicos dessa linguagem como :- e ?- foram considerados novas classes de *tokens*, para facilitar a sua identificação.

A Figura 2 representa o transdutor utilizado para reconhecer os *tokens* da linguagem.

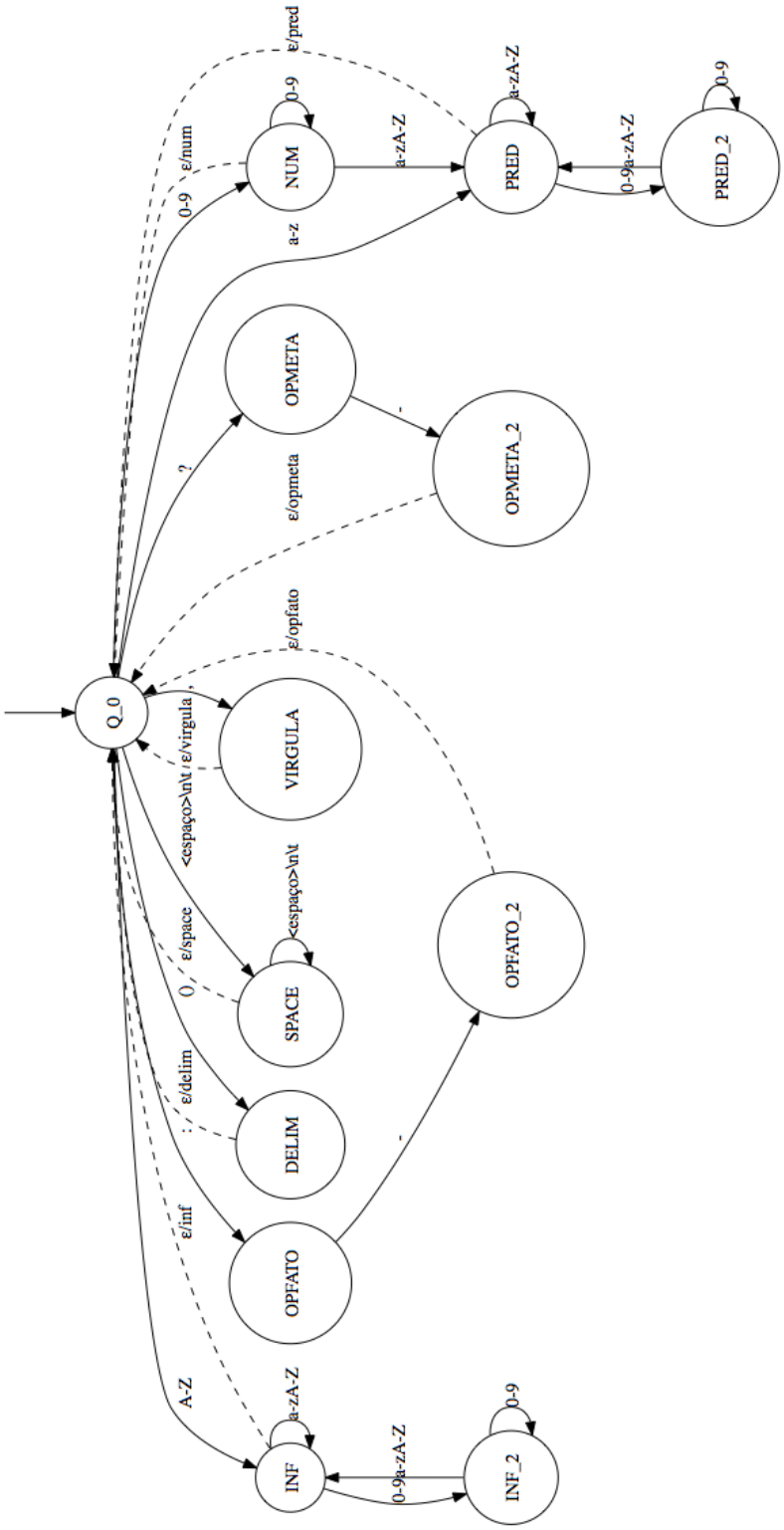

Figura 2 – Transdutor desenvolvido

4 Analisador sintático

A criação do analisador sintático, assim como com o analisador léxico, ocorreu antes da segunda prova e utilizou parte da estrutura criada para o trabalho da disciplina.

A título de recordação, 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. Com isso em mente, convertemos a sintaxe da linguagem *SimpPro* para WIRTH, como visto abaixo.

```
1 PROGRAMA = FATOS CLAUSULAS METAS.
2 FATOS = "(" FATO ")" { "(" FATO ")" }.
3 CLAUSULAS = "(" CLAUSULA ")" { "(" CLAUSULA ")" }.
4 METAS = "(" "?-" PRED DADO ")".
5 FATO = PRED ":-" DADO.
6 DADO = ( PRED | NUM ) { "," ( PRED | NUM ) }.
7 CLAUSULA = PRED ARGS ":-" ( "(" LCLAUSULA ")" |  DADO ).
8 LCLAUSULA = [ "not" ] "(" INF DADO ")" { ( "&" | "or" ) [ "not" ] "("
     INF DADO ")" }.
9 ARGS = ( INF | PRED | NUM ) { "," ( INF | PRED | NUM ) }.
```

A partir do WIRTH acima, reduzimos a sintaxe para conter somente um autômato, como mostrado abaixo.

```
1 PROGRAM = "(" PRED ":-" ( PRED | NUM ) { "," ( PRED | NUM ) } ")" { "("
     PRED ":-" ( PRED | NUM ) { "," ( PRED | NUM ) } ")" } "(" PRED ( INF
     | PRED | NUM ) { "," ( INF | PRED | NUM ) } ":-" ( "(" [ "not" ] "("
     PRED ( INF | PRED | NUM ) { "," ( INF | PRED | NUM ) } ")" { ( "&" |
     "or" ) [ "not" ] "(" PRED ( INF | PRED | NUM ) { "," ( INF | PRED |
     NUM ) } ")" } ")" |  ( PRED | NUM ) { "," ( PRED | NUM ) } ) ")" {
     "(" PRED ( INF | PRED | NUM ) { "," ( INF | PRED | NUM ) } ":-" ( "("
      [ "not" ] "(" PRED ( INF | PRED | NUM ) { "," ( INF | PRED | NUM ) }
      ")" { ( "&" | "or" ) [ "not" ] "(" PRED ( INF | PRED | NUM ) { "," (
      INF | PRED | NUM ) } ")" } ")" |  ( PRED | NUM ) { "," ( PRED | NUM
     ) } ) ")" } "(" "?-" PRED ( PRED | NUM ) { "," ( PRED | NUM ) } ")".
```

Através de um script, o WIRTH gerado foi então submetido ao site do Hugo Baraúna, seu resultado salvo em arquivos locais, o JFLAP aberto automaticamente e a figura do autômato armazenada localmente, além de gerar automaticamente o pdf impresso para a primeira parte da segunda prova. A Figura 3 mostra o autômato final.

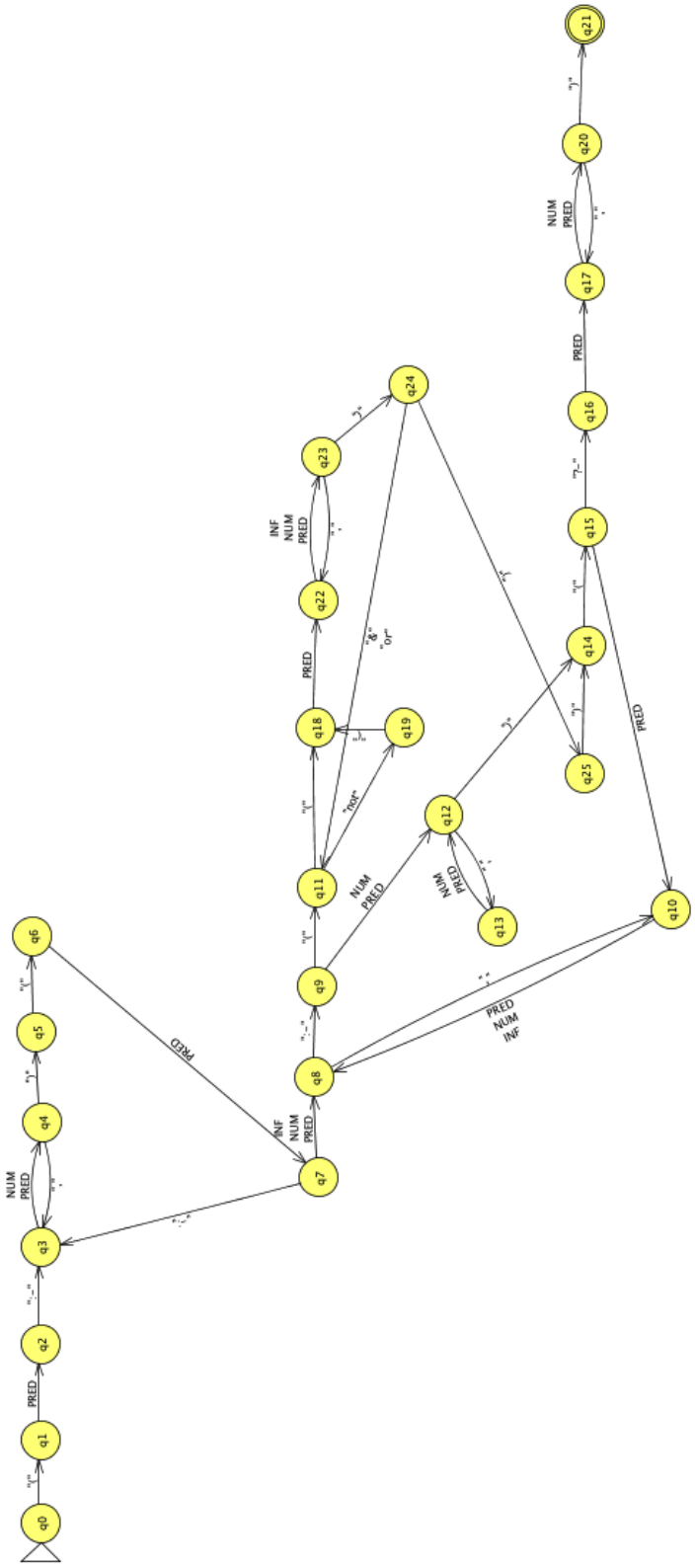

Figura 3 – Autômato *Program*

5 Analisador semântico

Durante a parte de especificação da segunda prova, a etapa 2, escolhi especificar toda a estrutura da linguagem *SimpPro* dentro da memória do *RNA*, com a inferência sendo executada pelo interpretador *RNA*. Porém, ao observar que a variável **strg** só possuía 1 Byte, só seria possível acessar 256 células de memória, o que poderia complicar a implementação do motor de inferência no *RNA*.

A partir dessa constatação, decidi implementar a geração de novos fatos a partir dos fatos originais e das cláusulas no C, de forma progressiva, sem observar a meta desejada. Com a inferência completa, foi adicionado a fita do *RNA* somente a meta e a base de fatos completa, além do código que busca a meta na lista de fatos e retorna se encontrou ou não.

Foram criadas 11 ações semânticas, adicionadas no autômato em diferentes posições e exibidas na Figura 4.

A breve descrição das ações semânticas implementadas está listada abaixo:

1. **AS 1**: Insere a constante (PRED ou NUM) na lista de constantes.
2. **AS 2**: Insere a variável (INF) na lista de variáveis.
3. **AS 3**: Insere o fato na lista de fatos.
4. **AS 4**: Insere a parte esquerda da cláusula na lista de sentenças e adiciona a referência da sentença na cláusula a ser processada.
5. **AS 5**: Insere o operador lido (“&” ou “or”) na cláusula a ser processada.
6. **AS 6**: Insere o operador not na cláusula a ser processada.
7. **AS 7**: Ação que trata a operação (avo joao, maria :- jose, joaquim, 3), cláusulas que tem dados na parte direita da sentença. Como acordado com o professor, não será implementado por não ter correspondência com o resultado que é verdadeiro ou falso.
8. **AS 8**: Insere a cláusula já construída nas outras ações na lista de cláusulas.
9. **AS 9**: Insere a sentença lida como meta. Também chama o motor de inferência que gera os fatos inferidos a partir das cláusulas e dos fatos originais.
10. **AS 10**: Insere a sentença formada previamente na lista de sentenças e adiciona a referência da sentença na cláusula a ser processada.

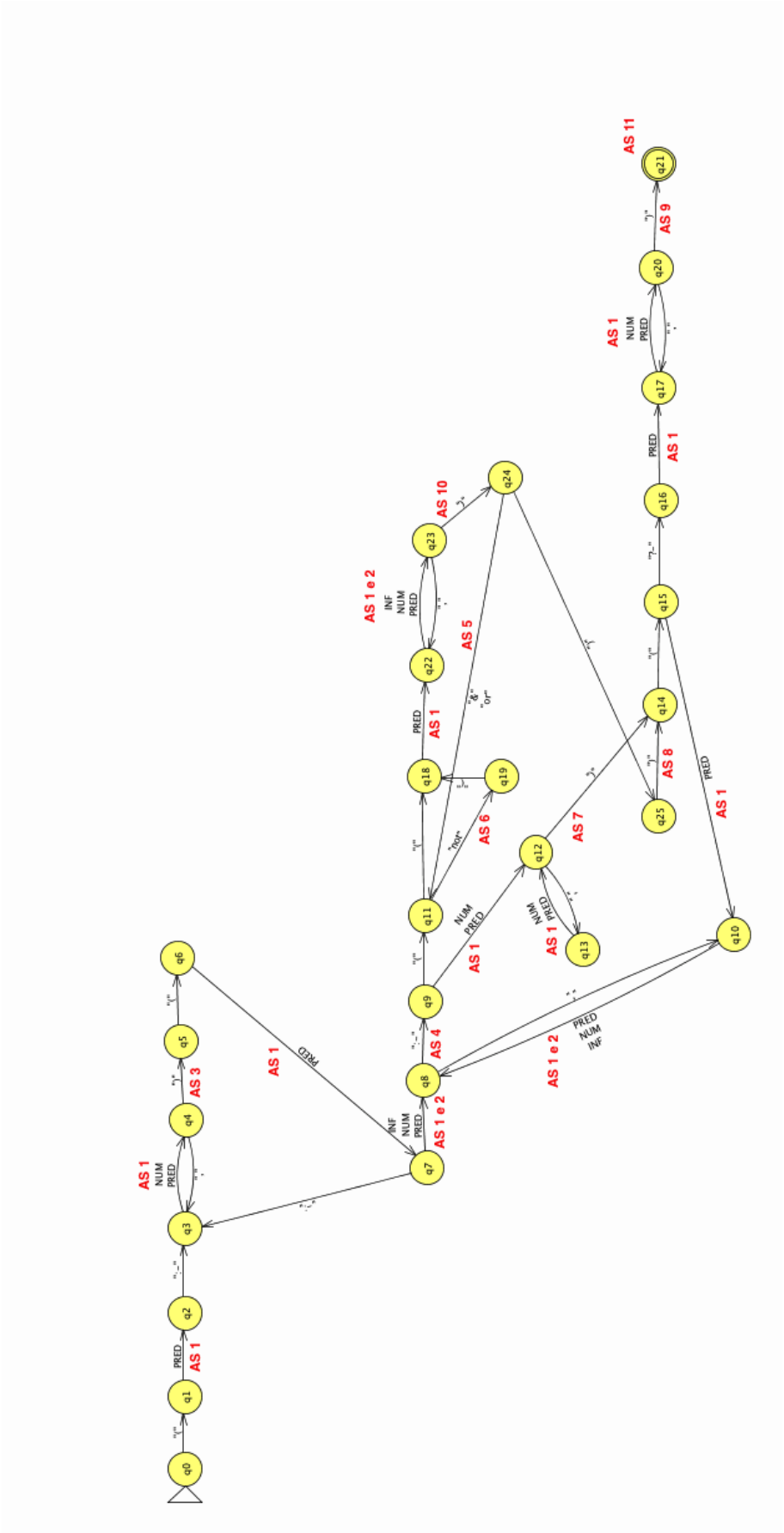

Figura 4 – Autômato *Program* com ações semânticas

11. **AS 11**: Ao terminar de ler o programa corretamente, chama a geração de código que escreve em *RNA* a meta, os fatos gerados e o código necessário para verificar se a meta está contida na lista de fatos e imprimir 1 para verdadeiro ou 0 para falso.

A geração de código foi efetuada através de uma estratégia que facilitou bastante o processo de criação dos algoritmos em *RNA*. Eu desenvolvi um conversor em C++ de C para RNA e de RNA para C, utilizando a descrição dos comandos disponibilizada na Wiki da linguagem e mostrado no Capítulo 2. O conversor está presente no código anexado, na pasta "rna/testes/conversor.cpp". Com esse conversor, desenvolvi os códigos a serem gerados em C e converti ao final para *RNA*, copiando o código gerado dentro do compilador. Cabe ressaltar que o conversor também foi crucial para entender e *debugar* o código *RNA* gerado durante o desenvolvimento.

6 Ambiente de execução e geração de código

Para executar o código *RNA* produzido pelo compilador, não foi necessário criar nenhuma função extra do lado do interpretador, somente fazer a correção do que não funcionava, como mencionado no Capítulo 2. O código gerado em *RNA* realiza os seguintes passos:

- Organiza a memória da seguinte forma:
  - Posição 0: 1
  - Posição 1: 0
  - Posição 2: posição inicializada com zero, reservada para a meta.
  - Posição 3: posição inicializada com zero, reservada para o resultado do while, se continua a buscar a meta na lista de fatos ou não.
  - Posição 4: posição inicializada com zero, reservada para saber se a meta foi encontrada ou não na lista de fatos. É a célula que retorna o resultado.
  - Posição 5: posicão inicializada com zero, reservada para saber se a lista de fatos já foi inspecionada completamente.
  - Posição 6: posição inicializada com a posição anterior a lista de fatos, reservada para ser o iterador que guarda o índice da lista de fatos que está sendo analisado.
  - Posições 7..7+(f-1): sendo f o número de fatos da lista de fatos, guarda a lista de fatos.
  - Posição 8+(f-1): sendo f o número de fatos da lista de fatos, sinaliza com o valor 0 que a lista de fatos acabou.
- Adiciona a meta.
- Adiciona a lista de fatos.
- Imprime o valor das variáveis antes da busca (para facilitar o *debug*).
- Adiciona o código de busca que funciona com um loop que a cada rodada:
  - Incrementa o iterador da posição 6.
  - Verifica se o valor inspecionado da lista de fatos é igual a 0 (o que significa que a lista já foi toda percorrida) e coloca o resultado da verificação na posição 5.

  - Verifica se o valor inspecionado da lista de fatos é igual a meta e coloca a o resultado da verificação na posição 4.
  - Coloca o valor da expressão `continue = !found && !is_over` na posição 3, para ser usada como expressão de continuação do while.

- Ao sair do while, imprime os valores da memória depois da busca (para facilitar o *debug*).

- Imprime o resultado do programa, 0 caso a meta não possa ser inferida da lista de fatos e cláusulas, 1 para o caso positivo.

7 Testes e exemplos de execução

Para realizar o teste do compilador e verificar sua execução, foi criado um arquivo README, adicionado ao código, que explica como testar o léxico somente, o compilador completo e o compilador com a execução no interpretador *RNA*. Para cada um deles, basta rodar **make clean** e **make <comando>**, <comando> = lextest, compilertest e runrna. Todos executarão tendo como base o arquivo examples/simprolog.pro.

Também cabe relembrar que deve ser utilizada a versão do interpretador *RNA* que está anexada ao código, com as mudanças propostas no Capítulo 2.

Segue abaixo dois exemplos de programas executados e seus respectivos resultados.

7.1 Programa 1

Abaixo, segue um exemplo de programa rodado:

```
1  (irmao :- joao, maria)
2  (pai :- jose, maria)
3  (pai :- mario, jose)
4  (mae :- joana, maria)
5  (pai :- joaquim, joana)
6
7  (pai X,Y :- ((pai X, Z) & (irmao Z, Y)))
8  (pai X,Y :- ((pai X, Z) & (irmao Y, Z)))
9
10 (avo X,Y :- ((pai X, Z) & (pai Z,Y) or (pai X, Z) & (mae Z, Y)))
11
12 (?- avo mario, joao)
```

O resultado do código executado pelo interpretador *RNA* segue abaixo:

```
Memoria antes (a partir da posicao 1): 0=000612345>=?@0
Memoria depois (a partir da posicao 1): 0=011=12345>=?@0
Resultado da query: 1
```

7.2 Programa 2

Abaixo, segue um exemplo de programa rodado:

```
1  (irmao :- joao, maria)
2  (pai :- jose, maria)
3  (pai :- mario, jose)
4  (mae :- joana, maria)
5  (pai :- joaquim, joana)
6
7  (pai X,Y :- ((pai X, Z) & (irmao Z, Y)))
```

```
8  (pai X,Y :- ((pai X, Z) & (irmao Y, Z)))
9
10 (avo X,Y :- ((pai X, Z) & (pai Z,Y) or (pai X, Z) & (mae Z, Y)))
11
12 (?- avo mario, joaquim)
```

O resultado do código executado pelo interpretador *RNA* segue abaixo:

```
Memoria antes (a partir da posicao 1): 0=000612345>?@A0
Memoria depois (a partir da posicao 1): 0=000@12345>?@A0
Resultado da query: 0
```