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

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.

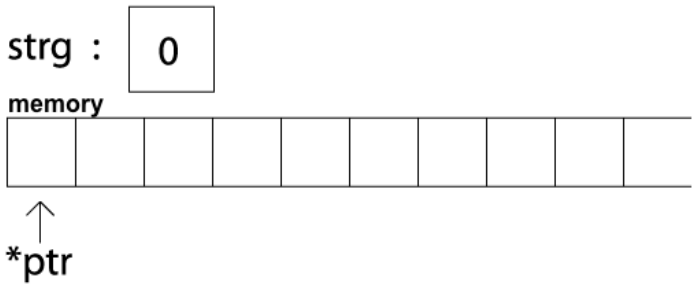

Figura 1 – Ilustração das estruturas de armazenamento do *RNA*.

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

- Explicação que foi utilizado o analisador léxico do trabalho, porém foram feitas adapatações, por exemplo, PRED vs INF e adicionado os operadores das cláusulas e meta como classe de tokens.
- Image transdutor léxico: images/transdutor.png

4 Analisador sintático

- Explicar que parte da estrutura criada para o trabalho que automatiza (script) a realização do sintático também foi aproveitada, permitindo apenas inserir um novo wirth para gerar os automatos, as respectivas imagens e até o pdf com as principais informações para a parte presencial (etapa 2) da P2
- A sintaxe em BNF fornecida para o *SimpPro* foi convertida para WIRTH (files/WIRTH_orig.txt) e, em seguida, reduzida a uma só máquina (files/WIRTH.txt).
- Através do JFLAP, criamos uma imagem que ilustra a única máquina *PROGRAM* (images/automato.png) utilizada para reconhecer a linguagem de entrada.

5 Analisador semântico

- ações semanticas
- usar images/semantico.png
- Interpretador corrigido
- geração de código (citar conversor de RNA para C e o inverso facilitaram muito o desenvolvimento e *debug* durante essa etapa).

6 Realização de testes

- falar sobre como testar (README e make) e como eu testei: teste compilertest e runrna
- relembrar a alteração no interpretador

7 Exemplo de execução

- files/simprolog.pro
- exemplos de execução do lextest, compilertest e runrna