C-minus com operações matriciais

Leonardo Maffei da Silva*

2019

Resumo

Este documento atende os fins de documentação da terceira parte do projeto final da disciplina *Tradutores*, ministrada pela professora Dr.a Cláudia Nalon, no segundo semestre de 2019, na Universidade de Brasília. Tal artefato descreve um pouco da implementação do analisador sintático, dificuldades encontradas durante tal processo; a nova gramática proposta em virtude de utilização de fonte anônima para construção das gramáticas anteriores; políticas de tratamento de erros e arquivos de teste para o analisador produzido.

Palavras-chave: C, linguagem, matriz, primitiva.

1 Introdução

Aqui é o lugar para fazer a descrição sumária da sua linguagem.

Implementar-se-á, até a versão final deste artigo, um compilador para a linguagem proposta. Para sua realização, serão utilizados os conhecimentos adquiridos na disciplina *Tradutores*, ministrada pela professora Cláudia Nalon.

2 Usuário característico

Destina-se ao estudante de álgebra linear, o qual pode usar a linguagem para, por exmeplo, confirmar se sua resolução de um sisema linear encontra-se correta, tudo isso de maneira rápida e eficiente.

Use o corretor ortográfico do seu sistema de edição favorito.

3 Motivação

Durante a realização do curso de Cálculo Numérico, o grupo do autor notou a ausência dessa *feature* na linguagem C. Desse modo, foi necessária a simulação desse tipo de dados, à época implementada por meio de inúmeras funções. Se houvese um tipo nativo para matriz bem como operações elementares sobre seus elementos, teria sido de grande auxílio à codificação dos diversos métodos numéricos requeridos pela disciplina.

*leoitu22hotmail.com@gmail.com. <https://www.linkedin.com/in/leonardo-maffei-ti/>

Não use termos em inglês quando eles são absolutamente desnecessários.

4 Gramática

Seja mais claro quanto à alteração. Foram retirados elementos da linguagem? Ou foi só mesmo a reformulação do conjunto de regras? E por que isto foi necessário?

Para esta entrega a gramática foi completamente reformulada. Sua especificação encontra-se abaixo:

Eu prefiro que me entregue a gramática sem anotações semânticas. O código é submetido junto com o restante da implementação.

```
program : global-stmt-list
global-stmt-list : global-stmt-list global-stmt
global-stmt-list : global-stmt
global-stmt : decl-fun
global-stmt : def-fun error
global-stmt : def-fun
global-stmt : decl-var ;
global-stmt : decl-var error
global-stmt : attr-var ;
global-stmt : block
def-fun : base-type ID ( param-list-void ) block
decl-fun : ahead base-type ID ( param-list-void )  ;
decl-var : mat base-type ID [num ] [num ]
decl-var : base-type id-arr
id-arr : ID [num-id ]
id-arr : ID
attr-var : mat-attr
attr-var : index-attr
attr-var : simple-attr
simple-attr : ID ATTR expr ;
index-attr : ID [num-id ] ATTR expr ;
mat-attr : ID ATTR [num-list-list ]
mat-attr : ID [num-id ] ATTR { num-list }
mat-attr : ID [num-id ] [num-id ] ATTR expr
num-list-list :  num-list-list { num-list }
num-list-list : num-list
num-list : num-list num
num-list : num
num-list : ID
stmt : return expr ;
stmt : COPY ( ID ID  ) ;
stmt : READ ( ID [num-id ] [num-id ] ) ;
stmt : READ ( ID [num-id ] ) ;
stmt : READ ( ID ) ;
stmt : PRINT expr ;
stmt : call ;
stmt : decl-var ;
stmt : attr-var ;
stmt : flow-control
stmt : loop
param-list-void : void
param-list-void : param-list
param-list : param-list param
param-list : param
```

```
param : base-type ID
param : mat base-type ID
loop : while ( expr ) block
flow-control : if ( expr ) block else flow-control
flow-control : if ( expr ) block else block
flow-control : if ( error ) block else block
flow-control : if ( expr error block else block
block : { stmt-list }
block : { }
stmt-list : stmt-list stmt
stmt-list : stmt
expr : add-expr relop add-expr
expr : add-expr
relop : <=
relop : <=
relop : !=
relop : ==
relop : >
relop : <
add-expr : add-expr addop term
add-expr : term
addop : +
addop : -
term : term mulop bin
term : bin
mulop : @
mulop : *
mulop : /
mulop : @@
bin : bin bin-logi unary
bin : unary
bin-logi : &&
bin-logi : ||
unary : unary-op factor
unary : factor
unary-op : !
unary-op : &
factor : ( expr )
factor : aux
factor : call
factor : 'ascii'
aux : ID [ expr ] [expr ]
aux : ID [ expr ]
aux : num-id
num-id : num
num-id : ID
call : ID ( arg-list )
call : ID ( void )
arg-list : arg-list , arg
```

```
arg-list : arg
arg : mat-arg
arg : aux
mat-arg : ID V_INT V_INT
ascii :  V_ASCII
base-type : CHAR_TYPE
base-type : INT_TYPE
base-type : FLOAT_TYPE
num : V_FLOAT
num : V_INT
V_FLOAT : [0-9]+[.][0-9]+
V_INT : [0-9]+
V_CHAR : [a-zA-Z0-9_]
ID : [a-zA-Z_][a-zA-Z0-9_]+
```

:-) Vê? Eu só fico sabendo aqui o que foi feito.

A nova gramática gera linguagem semelhante à anteriormente gerada, sendo que as principais diferenças ficam por conta da passagem de matriz como parametro, adição da palavra reservada **void** para quando se for declarar uma função sem parâmetro algum, remoção de alguns operadores (por exemplo, o '-' unário) e adição de alguns erros de provável ocorrência (por exemplo, o *não fechamento de parêntese* do **if**).

As palavras reservadas da linguagem são: **void**, **if**, **else**, **while**, **int**, **float**, **char**, **return**, **mat**, **ahead**, **READ** e **PRINT**. **void** deve ser usado quando declarando/definf/chamando uma função que não recebe parâmetro algum; **mat** indica a declaração de um dado do tipo matriz; **ahead** é utilizado para declaração de função; **READ** e **PRINT** são os comandos da linguagem para leitura e escrita.

Eu não acho que essa construção seja necessária...

5 Semântica

A semântica da linguagem é semelhante à da linguagem C: declarações de variáveis (a menos das do tipo **mat**), funções e expressões têm semântica similar. Sendo esta linguagem uma extensão de um subconjunto da linguagem *C*, a principal diferença está no tipo de dados **mat** (abreviação de *matrix*). Esse tipo de dados é similar aos *arrays* em C, porém limitado do ponto de vista da composição pois não é possível a criação de matrizes aninhadas, nem de matrizes de vetores. Entretando, é possível a realização das quatro operações aritméticas básicas diretamente com matrizes, bem como a realização de potenciação de matrizes de forma *rápida* e algumas operações sobre elas, como resolução de sistemas lineares e escalonamento. Multiplicação e potenciação de matrizes são respectivamente expressas pelos novos operadores @ e @@. Finalmente, a linguagem apena aceita passagem de parâmetros po referência e toda variável é indexável.

6 Exemplo de programas na linguagem

A seguir, trechos de código pertencente à nova linguagem.

```
1 int main() {
2 float a = 10.1;
3 float c = 10.;
4
5 mat int m[3][3] = [
6   {1, 0, 0}
```

```
  {0 ,1, 0}
  , 0, 1}
];
READ(a);
}'a'; '\n'; '\r'; '\\';
```

```
float main;
{1,2,3};
"Pode ir, tudo bem..."
/* Lucero mto bom */
print('h');
print('e');
print('l');
print('l');
```

7 Exemplo de programa não pertencentes à linguagem

```
" string sem dim!

int main() $$ {
;;;
}
```

```
float chr(void);
'\\\\';
/*
int main() {

}
print("e agora, joseh?")
Obs: a exemplo do tratamento proporcionado pelo gcc,
nao eh feito nenhuma tentativa de recuperacao para
erros de comentarios sem fechamento ;)
```

8 Implementação

Reescreve isto.

O presente analisador necessita, além dos arquivos citados abaixo léxico tem como dependências, além do programa *flex*, os respectivos arquivos fonte, além de seus respectivos cabeçalhos (com exceção do último, por tratar do código fonte utilizado pelo *flex* para geração do arquivo **lex.yy.c**):

- ShowTree.c
- Function.c
- Prints.c
- grammay.y
- lexico.l

Listing 1 – bash version

```
make clean
flex leo.l
make
./lexico <caminho-para-arquivo>
```

8.1 Sintático

Parece que está faltando alguma coisa aqui...

8.2 Funcionamento

8.2.1 Léxico

Após gerado o analisador léxico seguindo os passos descritos no início desta seção, sua utilização é bastante simples: execute o programa passando como argumento o caminho para o arquivo que deve ser aberto e processado pelo léxico. O programa então lê sequenciamente o arquivo caractere a caractere e vai exibindo os *tokens* presentes no arquivo apontado à medida que os encontra, bem como trata os erros descritos na subseção logo abaixo. A saída é exibida na saída padrão (usualmente, um console) e de forma *colorida*. Entende-se que isso facilita a visualização por parte dos humanod; contudo, isso deve ser adaptado na próxima fase para que seja gerado um arquivo estruturado contendo todos os tokens identificados, o qual será utilizado pelo analisador sintático.

Não é assim que funciona.

8.3 Tratamento de Erros

8.3.1 Léxico

O analisador léxico desenvolvido é capaz de detectar os seguintes erros:

1. *string* sem fechamento
2. comentários em bloco sem fechamento
3. caracteres não pertencentes à linguagem (individualmente)

gerar referências adequadamente

O erro **??** é tratado considerando que o usuário termina a *string* ao final da linha, visto que não são permitidas *strings* multi-linhas, e continuando a análise como se não houvesse erro. O próximo erro (**??**) não é exatamente tratado; a abordagem utilizada é a mesma do compilador gcc 7.7.0: é emitido um aviso ao usuário informando-lhe linha e coluna onde se inicia o comentário não finalizado. Por fim, o último erro (**??**) é tratado informando ao usuário as ocorrências desses caracteres, porém sem entrar em modo de pânico.

8.3.2 Sintático

O analisador produzido reporta alguns erros comuns, quais sejam:

- não inserção de ponto e vírgula após o valor a ser retornado pelo *return* e o não fechamento de parêntese
- expressão vazia (não são permitidas expressões lugal algum)
- não inserção de parêntese direito relativo à condição do *if*

Em caso de outros erros, a análise simplesmente é abortada. Em caso dos erros acima, a árvore sintática é montada naturalmente pelo *bison*.

Bison

Não é não. É você que tem que construir a árvore.
Melhore a descrição da sua implementação.

8.4 Dificuldades Encontradas

8.4.1 Léxico

Uma das dificuldades foi sem dúvida a familiarização com a ferramente *flex*, a qual demonstrou-se muito competente porém não tão intuitiva, bem como alguns trechos de seu manual. Além disso, surgiram dúvidas a respeito do que seriam erros léxicos, como por exemplo qual o escopo dessa categoria de erros.

Além das dúvidas acima, foram encontrados obstáculos na automação da geração do analisador por meio da ferramenta *make*, questões sobre como deveria ser exibida a sequência de *tokens* (quais possíveis metadados deveriam ser exebidos?) e algumas pontualidades acerca de recursão na gramática.

Em termos de codificação, além do problema recém-citado, não foi possível implementar a tabela de símbolos a tempo da entrega (ocorreram problemas de vazamento de memória e alocação equivocada). Embora esta não seja fundamental para a exibição dos tokens lidos em sequência, sua presença certamente enriqueceria o presente trabalho.

8.4.2 Sintático

Este trabalho exigiu esforço de codificação e pesquisa muito acima do esperado. As dificuldades principais sao listadas a seguir:

- integração entre bison e flex, em especial relativo ao entendimento das interações entre as seções e de suas relevâncias
- decidir quais atributos deveriam estar na tabela de símbolos
- exibição da árvore (não foi possível fazê-lo; em algum ponto ocorre falha de segmentação)
- projeto da gramática: a presente gramática foi feita do zero; contudo, contém ainda alguns erros e possui um número absurdo de regras, em relação às gramáticas de muitos dos outros aluno
- volume de codificação ascrescido de repetição: muito do código feito segue um padrão muito bem definido, basicamente um modelo. Digitar tudo isso foi bastante tedioso e não parece ter contribuído para a geração de conhecimento relativo à teoria abordada nas aulas ou sobre a prática da implementação de compiladores
- passagem de dados do *flex* para o *bison*

O principal problema encontrado, no sentido de ter tomado muito tempo de codificação, foi a questão da árvore sintática. Tentou-se fazer uso das variáveis utilizadas pelo *bison* para, ao final da análise sintática, gerar a imagem da árvore de derivação. Entretando, erros acometidos durante a fase de codificação ou estruturação da solução para exibir a árvore impossibilitaram sua realização. Tal impossibilidade deu-se por conta de falha de segmentação a qual ocorre em uma das primeiras funções chamadas para exibir a árvore de *parsing* **após sua completa montagem (implícita)**.

A construção da árvore foi feita por meio do padrão

```
1
2 head : body {
3   $$ = make_Head($1);
4 }
```

onde para cada corpo de regra há uma função *make_<NOME>* associada. Há também um tipo de dados específico para cada cabeça de regra, bem como cada tipo desses possui uma union a qual armazena structs cujo conteúdo são outros dados associados aos tipos de cada nó. Por exemplo:

```
1
2 head : body {
3 $$ = make_Head($1);
4 }
```

8.5 Arquivos de teste

Os arquivos de teste encontram-se na pasta *test*. Os arquivos cujo texto pertence (ou deveria pertencer) à linguagem gerada pela gramática estão prefixados pela letra **c** e os que não pertencem à ela são prefixados pela letra **e**. Há também arquivos de cujo nome é prefixado com a letra **t**, os quais representam sequências não pertencentes à linguagem porém cuja análise sintática foi bem sucedida graças às políticas de tratamento de erros implementada.

9 Referências

Foram utilizados basicamente os manuais do flex (**??**) e do bison (**??**). Tais fontes não eram de tão fácil compreensão, o que demandou esforços consideráveis em algumas situações.

Incluir seção com bibliografia.