

C-mat: um mini-C com matrizes

Leonardo Maffei da Silva*

2019

Resumo

Este documento atende os fins de documentação da quarta 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 semântico, bem como dificuldades encontradas durante tal processo. Para esta fase novamente refez-se a gramática, a fim de reduzir a quantidade de regras e tratar alguns *bugs* presentes na antiga, aproveitando o fato de que seria necessário a reimplementação completa do analisador sintático.

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

1 Introdução

Implementar-se-á, até a versão final deste artigo, um compilador para a linguagem C-mat, a qual consiste essencialmente de uma *extensão para um subconjunto da linguagem C*. C-mat permite a utilização de matrizes 2-D enquanto tipos de dados nativos da linguagem, sendo possível a realização de operações comumente realizadas quando são feitas operações entre matrizes (multiplicação, adição, subtração, adição de escalar, potenciação, etc). Para a implementação, 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 exemplo, confirmar se sua resolução de um sistema linear encontra-se correta, tudo isso de maneira rápida e eficiente.

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

3 Motivação

Durante a realização do curso de Cálculo Numérico, o grupo do autor notou a ausência dessa funcionalidade na linguagem C. Desse modo, foi necessária a simulação desse tipo de dados, à época implementada por meio de inúmeras funções. Se houvesse 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.

4 Gramática

Para esta fase, a diferença mais marcante na gramática está na especificação de precedência de operadores não mais inerente à gramática (conforme fora feito nos trabalhos anteriores), mas sim por meio das diretivas *%left* e *%right* do *bison*. Além disso, novas *features* foram adicionadas à gramática, a saber:

- declaração e inicialização de variáveis em apenas um *statement*
- não utilização da palavra reservada **void** quando se define ou chama função sem parâmetros
- pode-se ter ; como *statement* local (desse modo, mais de um ponto e vírgula não causa erro)

Foram também retiradas algumas coisas, quais sejam:

- remoção dos tipo **char** e **void**
- remoção de regras que serviam como "demultiplexadoras dummy" de operadores
- declaração de variáveis deve vir antes dos demais *statements* de uma função

O tipo **char** foi removido pois sua única função reside em servir de parâmetro ao comando **PRINT**. Agora, não existe mais esse tipo de dado, mas é possível a criação de *statements* utilizando diretamente o caractere desejado (apenas letras e números da tabela *ascii*); por exemplo, **PRINT('a')**. Vale ressaltar que para a impressão de quebra de linha, deve-se invocar o comando **PRINT** passando-lhe como "argumento" o "caractere vazio"; ou seja, deve-se ter **PRINT("")**.

Além das adições / remoções enumeradas, foi adicionada uma regra específica para declaração de variáveis, separada da declaração de parâmetros, além da possibilidade de explicitar que um parâmetro é um vetor. A seguir, a especificação da nova gramática:

```
program : globalStmtList
globalStmtList : globalStmtList globalStmt
|
globalStmt : defFun
| declFun ;
| declOrdeclInitVar
| declFun error
declFun : ahead BASE_TYPE ID
```

```

typeAndNameSign : BASE_TYPE ID
| BASE_TYPE ID [ num ]
| mat BASE_TYPE ID [ num ] [ num ]
declOrdeclInitVar : typeAndNameSign ;
| typeAndNameSign = rvalue ;
| typeAndNameSign = rvalue error
paramListVoid : paramList
|
paramList : paramList , param
| param
param : BASE_TYPE ID
| BASE_TYPE ID [ ]
| mat BASE_TYPE ID
localStmtList : localStmtList localStmt
|
localStmt : call ;
| lvalue = rvalue ;
| flowControl
| loop
| return expr ;
| return expr error
| IREAD ( lvalue ) ;
| FREAD ( lvalue ) ;
| PRINT ( rvalue ) ;
| PRINT ( V_ASCII ) ;
| COPY ( lvalue lvalue ) ;
| ;
flowControl : if ( expr ) block else flowControl
| if ( expr ) block else block
| if ( expr ) block
| if ( expr error block else block
| if error expr ) block else block
| if ( error ) block
loop : while ( expr ) block
defFun : BASE_TYPE ID (
numListList : numListList { numList }
|
numList : numList , num
| num
block : { localStmtList }
declList : declList declOrdeclInitVar
|
expr : expr + expr
| expr - expr
| expr * expr
| expr / expr
| expr % expr
| expr @ expr
| expr @@ expr

```

```

| expr == expr
| expr != expr
| expr >= expr
| expr <= expr
| expr > expr
| expr < expr
| expr && expr
| expr || expr
| ! expr
| & expr
| ( expr )
| ICAST ( expr )
| FCAST ( expr )
| lvalue
| call
| num
call : ID ( argList )
| ID ( )
argList : argList , arg
| arg
arg : lvalue
| ID ( expr ) ( expr )
num : V_INT
| V_FLOAT
lvalue : ID
| ID [ expr ]
| ID [ expr ] [ expr ]
rvalue : expr
| { numList }
| [ numListList ]

```

As palavras reservadas da linguagem aparecem em **negrito** na gramática, os outros *tokens* estão em letras maiúsculas e as regras de produção estão em letras minúsculas (com eventuais letras maiúsculas entre as demais para facilitar a legibilidade).

As principais limitações impostas pela gramática de C-mat são listadas abaixo:

- declaração de variáveis apenas no escopo global ou no início de funções
- matrizes são sempre bidimensionais
- vetores são unidimensionais e suportam apenas os tipos **int** e **float**

Essas limitações tem por objetivo simplificar a implementação do tradutor dado o período de apenas um semestre e, por serem garantidas pela gramática, não requerem análise semântica.

5 Semântica

A principal mudança nessa nova fase do projeto está na impossibilidade de indexação de variáveis que não são vetores nem matrizes (ideia inicial do projeto). De resto,

não houve grandes alterações. A semântica é em grande parte idêntica à da linguagem C, em especial nos pontos que possuem em comum (operações de adição, subtração, etc) e nos tipos básicos *int* e *float*.

Construtos de controle de fluxo se comportam de maneira idêntica à de C, embora seja necessário que todo *statement if/else* seja seguido por um bloco de código. O construto *while* tem funcionamento idêntico em ambas as linguagens; operações de *cast* devem ser feitas por meio dos operadores *ICAST* e *FCAST*, mas caso não sejam utilizados e conversão ainda pode ser feita, embora seja emitido mensagem de aviso ao usuário.

Só é possível a realização de *cast* entre escalares (**int** e **float**); outras tentativas de conversão são caracterizadas como erros. C-mat possui apenas dois escopos: global e local. Caso uma variável utilizada dentro de uma função não tenha sido declarada, esta é buscada no escopo global. Caso não seja encontrada, é reportado erro. Não foi implementado o escopo de bloco com o único intuito de facilitar a implementação da análise semântica. Um dos motivos foi o débito relativo ao analisador sintático que para ser quitado requereu, conforme citado em 6.5.1, quantia significativa de tempo.

Operadores binários são de maneira geral associativos à esquerda, com exceção para o operador de potenciação de matriz **@@**, visto que a operação de potenciação é associativa à direita tal qual na matemática usual. Há operadores para as seguintes operações:

- adição/subtração
- multiplicação/divisão
- resto da divisão
- comparação (maior que, menor que, maior ou igual a, menor ou igual a, igual a e não igual a)
- AND e OR lógicos
- NOT lógico
- operador endereço (&)
- multiplicação de matrizes
- exponenciação de matrizes

Todas as operações acima, a menos das relacionadas a matrizes e do operador %, podem ser aplicadas a dois escalares quaisquer, embora caso esses operandos não sejam de mesmo tipo será mostrada mensagem de aviso ao usuário. Em caso de operação entre tipos distintos, é feita a conversão de tipos: inteiros são convertidos para ponto flutuante na maioria dos casos. Operadores relacionais são um caso especial; quando aparecem, o resultado da operação é sempre um **inteiro**: 0 ou 1. O operador **!** nada mais faz do que troca o valor de seu operando 0 para 1 ou de qualquer outro valor para 1. Em caso de o valor ser um ponto flutuante, a conversão para inteiro é feita, e então seu efeito é aplicado ao valor convertido. Em qualquer outro caso, considera-se que houve um erro.

É possível definir a assinatura de uma função antes de sua definição por meio da estrutura **ahead** `BASE_TYPE ID`; contudo, isso só pode ser feito uma vez para cada

nome de função (limitação que não limita a expressividade da linguagem, porém facilitou a implementação). Desse modo, pode-se efetuar checagem de tipo e de parâmetros de funções sem que seja necessário percorrer a árvore abstrata mais de uma vez, e portanto é possível a implementação de recursão indireta em C-mat .

Conforme entregas passadas, há vetor como tipo de dados composto; porém, não é possível ter vetores multidimensionais e só é possível a instanciação de vetores de escalares. Matrizes podem ser contruídas enquanto tipos nativos da linguagem, em notação similar ao que seria um vetor bidimensional, porém com o adicional de se necessário informar qual o tipo escalar que a matriz armazena. Contudo, novamente por conta de possíveis problemas na hora da tradução, C-mat implementa apenas *matrizes 2D*. Ainda assim, acredita-se que grande parte dos problemas encontrados pelo usuário a que se destina C-mat não lidam com matrizes com mais de duas dimensões e, portanto, essa limitação não deve ser um problema de maneira geral.

6 Implementação

Nesta fase houve pouca alteração no analisador léxico, pois o conjunto de tokens manteve-se praticamente o mesmo. As principais mudanças foram:

- remoção das funções de impressão
- total desconhecimento da tabela de símbolos: toda sua manipulação é realizada durante a análise semântica
- remoção das palavras reservadas **void** e **char**
- atribuição de matriz é feita como fosse um vetor com duas dimensões em C, com a diferença de que os sub-vetores do vetor maior não são separados por vírgula; basta deixar um ou mais espaços em branco entre eles

Já a gramática, bem como o analisador sintático, foram refeitos do zero, embora se tivesse em mente que a linguagem gerada pela gramática deveria ser bastante similar à que era gerada pela antiga. O analisador sintático teve de ser refeito pois a estrutura adotada anteriormente não era escalável, de difícil manutenção e possuía número bastante alto de estruturas de dados *dummy*.

A nova implementação trata cada nó da árvore abstrata como uma estrutura de dados que é comum a todos eles, diferenciando-se pelo nome do token e outros atributos, sendo o principal sua entrada na tabela de símbolos (cada nó da AST correspondente a algum identificador possui um ponteiro para a entrada na tabela de símbolos que contém informações sobre si).

A estrutura de árvore é obtida fazendo-se cada nó relativo a uma cabeça de produção apontar para o primeiro de seus filhos, o qual juntamente com seus irmãos encadeia a lista de filhos da cabeça da regra. Ações semânticas são realizadas quase sempre ao final da redução de uma regra; exceções a este padrão são encontradas na regras de declaração e definição de função bem como na regra que gera chamada de função, ações essas que setam a variável cujo valor é o escopo atual ou verificam se um identificador representa mesmo uma função.

Toda a análise semântica é realizada juntamente com a montagem da árvore abstrata por meio de *ações semânticas*, sem necessidade de percorrê-la mais de uma vez

(exceção para a checagem de assinatura de chamada e definição de função, que não exatamente percorre a árvore mais de uma vez, mas apenas a lista de parâmetros ou argumentos). Logo, é feita em *apenas uma passada*. Tal decisão foi tomada pois previu-se que a análise em mais de uma passada seria bastante trabalhosa e não traria benefícios significativos (o principal benefício seria a possibilidade de recursão indireta; porém, para resolução desse problema, C-mat permite a declaração de função antes de sua utilização).

6.1 Funcionalidades

Relata-se a seguir, por meio de enumeração, quais funcionalidades da análise semântica foram implementadas (total ou parcialmente):

- verificação de tipos
- verificação de escopo:
 - utilização de variável se e somente se já foi declarada
 - declaração de variável de mesmo nome sob mesmo escopo acusa erro (parâmetros são incluídos nesse caso)
- determinação do tipo de expressões
- verifica se chamada de função usar identificador de função de fato, e acusa erro caso seja nome de variável

6.1.1 Limitações

Primeiramente, ressalte-se que boa parte da análise semântica foi realizada com sucesso; contudo, há algumas regras cujas ações semânticas estão faltantes (por exemplo, em dois dos três corpos das regras de produção cuja cabeça é **typeAndNameSign**, não é feita a checagem de compatibilidade de tipo da declaração prévia do identificador. Entretanto, essa checagem a ser feita é análoga à realizada no primeiro corpo de **typeAndNameSign**).

Em segundo lugar, não estava sendo feita a checagem por ponteiro nulo em todos os locais possíveis. Inclusive, é por este motivo que declarações de variável de mesmo nome causam o *crash* da versão entregue do programa. Ao longo da confecção deste relatório, este bug foi corrigido, bem como outros como a ausência de ação semântica na para a regra **declOrdeclInitVar : typeAndNameSign ;**.

Também não foi possível consertar os vazamentos de memória da entrega passada, pois o analisador foi reimplementado. Quanto aos vazamentos de memória do novo analisador, estes não foram resolvidos a tempo. A memória relativa à árvore gerada pelo *bison* é em grande parte liberada, mas há vazamentos.

Como último *bug* detectado nessa fase do projeto, a dedução do tipo de expressão para o caso do operador de potenciação de matrizes estava errada. O problema residia na função **bin_expr_type** e foi corrigido durante a produção deste documento.

6.2 Novas funções

Conforme dito na seção 6, muito teve de ser refeito. Esta subseção apresenta as principais funções tanto do novo analisador sintático quanto do analisador semântico. As

funções utilizadas pelo analisador sintático são primariamente as disponíveis na biblioteca *Tree*, a qual implementa o tipo *No*, utilizado extensivamente pelo sintático para montagem da árvore. Tais funções são:

- *No** *No_New/No** *Token_New*: funções utilizadas para criação dos nós da AST
- *void add_Node_Child_If_Not_Null*: função para adicionar nós como filhos de um outro nó
- *void free_Lis/void free_all_child*: funções responsáveis por liberar a memória alocada durante a construção da AST
- *void print_res*: mostra as entradas da tabela de símbolos
- *void show_Sub_Tree/ void show_Lis*: mostra a árvore produzida durante a análise sintática

Na entrega passada, algumas funções relativas à análise semântica, especificamente a inserção de identificadores na tabela de símbolos, eram realizados por funções antes implementadas no mesmo arquivo que contém o código fonte do analisador sintático. Nesta entrega, as definições de tais funções foram movidas para uma biblioteca específica de funções a serem utilizadas na análise semântica. Desse modo, reduziu-se um pouco o tamanho do arquivo fonte do ASS. Tais funções encontram-se agora implementadas no arquivo *SemanticChecker.c*, cujo *header* é incluído pelo ASS.

Tais funções utilizadas para essa fase de análise semântica são, principalmente:

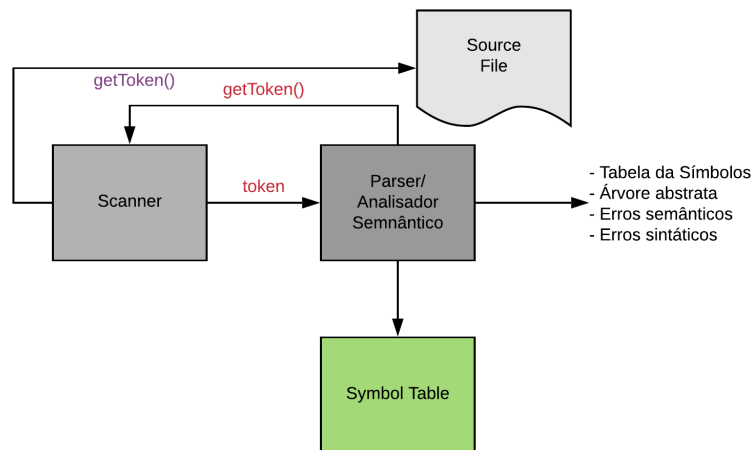
- *int match_paramList*: função responsável por checar se parâmetros de uma função batem com sua chamada ou com sua declaração prévia
- *void link_symentry_no*: função responsável por fazer uma entrada da tabela de símbolos apontar para um nó da AST e vice-versa
- *Type bin_expr_type*: retorna qual o tipo de uma expressão binária
- *SymEntry** *add_entry*: adiciona novo nome de identificador na tabela de símbolos, caso não exista registro prévio no escopo em questão. Caso contrário, acusa erro semântico e retorna *NULL*
- *SymEntry** *last_decl*: retorna ponteiro para a última declaração do identificador (declaração mais próxima). Útil para checar se identificador é usado incorretamente; por exemplo, chamada de função usando nome de variável.
- *SymEntry** *was_declared*: não mais usada. Renomeada para **last_decl**. Motivo foi dar mais semântica ao que a função faz.
- *void addToDel*: função utilizada para fazer o *tracking* das posições de memória utilizadas para alocação das entradas da tabela de símbolos

As funções recém listadas são usadas, ao menos uma delas, em basicamente todo local no qual aparece um identificador, seja para checar por sua existência da tabela de símbolos ou ainda, no caso de funções, se os tipos e ordem dos parâmetros estão corretos.

6.3 Funcionamento do programa

A figura 1 ilustra o fluxo básico de informações entre os módulos principais (*scanner* e analisador *sintático/semântico* (ASS)).

Figura 1 – Esquema de funcionamento das análises léxica, sintática e semântica



O programa começa pela execução do módulo de análise, o qual imediatamente chama o *scanner* como co-rotina, e este retorna um *token* para o ASS. Este por sua vez decide entre imediatamente pedir outro token, executar alguma redução ou realizar ações semânticas. É neste módulo que os identificadores são inseridos na tabela de símbolos e realizada a ligação entre os nós da árvore correspondentes a identificadores e suas respectivas entradas da tabela de símbolos. Este ciclo se repete até que o analisador atinja fim de arquivo ou ocorra algum erro durante a montagem da árvore sintática por conta do recebimento de token inesperado. Caso o arquivo seja sintaticamente correto, ao final do programa é exibida a árvore abstrata gerada a partir do código fonte. Caso haja erros/"imprecisões"semânticas (imprecisão seria por exemplo uma atribuição cujos tipos não batem), mensagens são exibidas também na saída padrão do console. Vale ressaltar que erros semânticos **não impedem a geração da árvore abstrata**, mas podem gerar **inconsistências** nela e na tabela de símbolos.

Ao final da execução do programa, são exibidas a árvore abstrata produzida e em seguida a tabela de símbolos.

6.4 Tratamento de Erros

6.4.1 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 como condição do construto *while*
- não inserção de parêntese direito relativo à condição do *if*

Em caso de outros erros, a análise simplesmente é abortada. Em caso de erro acima, prossegue-se com a montagem da árvore, porém esta apresentará inconsistências.

6.5 Dificuldades Encontradas

6.5.1 Sintático

Embora esta fase seja relativa ao analisador semântico, faz-se necessário breve relato das dificuldades encontradas no desenvolvimento do analisador sintático pois, conforme dito em 6, este foi completamente feito. Nesta nova implementação não houve muitas dificuldades, pois já se sabia como lidar com a maioria dos problemas surgidos.

Um dos obstáculos encontrados foi quanto à decisão sobre quais tokens ou cabeças de regras deveriam possuir seus próprios nós na árvore, visto que algumas produções são *dummy* e identificadores certamente requerem um nó para cada, além claro de uma conexão com sua respectiva entrada na tabela de símbolos. Foi também encontrada dificuldade a respeito de quais campos deveriam estar presente na estrutura de dados que forma cada nó da árvore abstrata e sobre quais regras podiam de fato ser tratadas como *dummy*.

Não houve problemas acerca da montagem da árvore de maneira geral, a menos do encadeamento da lista de parâmetros; durante certa etapa do desenvolvimento, houve dúvidas sobre como proceder no encadeamento da lista de parâmetros. Ao final, optou-se por cada parâmetro fazer parte de uma lista encadeada simples cujos elementos eram todos do mesmo tipo de dados da árvore, porém não faziam parte da estrutura da árvore como os outros.

6.5.2 Semântico

Certamente a maior dificuldade enfrentada nessa fase foi a não percepção de todos os possíveis erros semânticos passíveis de ocorrência. Essa situação causou a imprevisibilidade do tempo necessário à conclusão do projeto, bem como consecutivos tropeços por não ter implementado a estrutura *No* com isso em mente. Dito isso, algumas das dificuldades superadas ao longo da realização desta fase foram:

- resolução de escopo:
 - como assegurar que parâmetros tinham como contexto a função à que pertenciam?
 - diferenciação de escopos (em termos de eficiência, uma função *hash* seria a melhor opção; porém, a fim de simplificar a implementação, é feito por meio da comparação de caracteres)
- lista de parâmetros / argumentos: qual a melhor maneira de encadeá-los?
- lista de argumentos: como saber se uma função é chamada com argumentos cujos tipos batem com a assinatura da função?

Além dessas dificuldades relacionadas especificamente à análise semântica, registre-se que a maior parte do prazo de três semanas e 3 dias para esta entrega foi gasto refazendo a gramática, entendendo melhor o funcionamento do *bison* e criando as estruturas de dados a serem usadas na construção da AST. Também, fatores externos

à disciplina (entregas de outras disciplinas) influenciaram bastante negativamente a qualidade da entrega.

Não foi possível implementar *tudo* que a análise semântica deveria fazer; isso deve ser feito para a próxima entrega e relatado posteriormente.

6.6 Arquivos de teste

Os arquivos de teste encontram-se nos subdiretórios da pasta *test*. Cada subdiretório contém os arquivos de teste relativos a cada fase do trabalho (léxico, sintático e semântico). A seguir listam-se as pastas, seguidas de suas descrições:

- *lexico*: contém arquivos com prefixo *c* ou *e*, os quais estão respectivamente corretos ou errados para o léxico
- *sintatico*: organização idêntica à pasta *lexico*
- *semantico*: contém as subpastas *certos* e *erro*, as quais contém os arquivos que estão corretos e errados, respectivamente, segundo a análise semântica

7 Referências

Foram utilizados basicamente os manuais do flex (PROJECT, 2019) e do bison (THEBISONTEAM, 2019), além da documentação do cabeçalho *uthash* (HANSON,). Tais fontes não são de fácil compreensão (a menos da última), o que demandou esforços consideráveis em algumas situações.

Referências

HANSON, T. D. *uthash User Guide*: uthash user guide. Acessado: 18-11-2019. Disponível em: <<http://troydhanson.github.io/uthash/userguide.html>>. Citado na página 11.

PROJECT, T. F. 2019. Acessado: 18-11-2019. Disponível em: <<https://westes.github.io/flex/manual/>>. Citado na página 11.

THEBISONTEAM. 2019. Acessado: 18-11-2019. Disponível em: <<https://www.gnu.org/software/bison/manual/>>. Citado na página 11.