C-mat: um mini-C com matrizes

Leonardo Maffei da Silva*

2019

Resumo

Este documento atende os fins de documentação 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 de cada fase do compilador entregue como projeto final da disciplina, bem como dificuldades encontradas durante tal processo. Esta é a versão definitiva dos documentos entregues anteriormente, com destaque para a fase de *geração de código*.

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

1 Introdução

Este documento serve aos fins de documentação do trabalho desenvolvido pelo autor ao longo do semestre, ao passo que cursava a disciplina Tradutores. Conforme proposto no primeiro relatório, foram implementadas operações de adição, subtração e multiplicação entre matrizes. Promessas não cumpridas, *bugs* e desafios encontrados ao longo do desenvolvimento do compilador serão devidamente relatados ao longo das subseções relativas à implementação de cada etapa do projeto.

Para a implementação, foram utilizados conhecimentos adquiridos na disciplina *Tradutores*, ministrada pela professora Cláudia Nalon, além de fontes outras como *YouTube* e *Stack Overflow*. Finalmente, ressalte-se a importância do suporte provido pela professora, a qual não mediu esforços para ajudar cada aluno que lhe solicitava auxílio.

2 Breve descrição da linguagem implementada

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 e subtração). Matrizes podem ter cmo tipo base inteiros ou número ponto-flutuante. Matrizes são estruturas *homogêneas*, assim como os *arrays* da linguagem. A principal diferença prática entre matrizes e *arrays* em C-mat é que estes não podem ser utilizados em operações aritméticas ou matricias.

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

A menos do diferencial acima citado, C-mat é uma versão simplificada da consagrada linguagem C: possui mecanismos de controle de fluxo similar a C (if/else/while, porém com certas restrições), escopo global e de função (todo código interno a uma função encontra-se sob o mesmo escopo da função), declarações de variáveis globais e locais, declaração prévia de função, passagem de argumentos por cópia e por referência e outros. Todos esses pontos serão detalhados mais adiante.

3 Motivação e Usuário característico

C-mat foi proposta tendo como motivação o desejo de possuir matriz como tipo de dados nativos em C, principalmente após experiência do autor com a disciplina de Cálculo Numérico. Na ocasião, foi necessária a implementação de operação entre matrizes tais como: multipllicação, adição, subtração e operações sobre linhas. Tal experiência não foi nada agradável e, pensando nos alunos que programam em C e necessitam cursar a referida disciplina ou nos estudantes de álgebra linear (cuja necessidade de resolução de sistemas lineares é evidente), *C-mat* foi desenvolvida.

4 Gramática

Ao longo do projeto, diversas gramáticas foram propostas para a linguagem: a entregava os construtos de programação necessários, mas não foi aceita pela professora dada a anonimicidade da fonte. Desse modo, refez-se a gramática para a entrega do analisador sintático. Entretanto, isso ainda não foi suficiente: a nova gramática era mais extensa do que o desejado, e também possuíam *bugs*. Não bastasse isso, para a entrega do analisador semântico *todo* o analisador sintático foi refeito, baseado nessa nova gramática, assim como o modo de impressão da árvore abstrata.

Essa última versão apresentava de início menos regras em relação à anterior, sendo em geral de mais fácil manutenção, além de corrigir *bugs*. Para a fase final do projeto, as seguintes mudanças ocorreram na gramática:

1. remoção da regra *numListList*, sendo que o construto *numList* passa a ser usado para inicialização de matrizes em C-mat
2. inclusão da regra *dummy* countNewFlow
3. inclusão da regra *newFlowControl*
4. remoção das regras de operações bit-a-bit
5. adição do operador módulo (**%**)
6. possibilidade de passar constantes escalares como argumentos de funções (anteriormente, era necessário salvar tal valor em uma variável e passá-la por cópia)

A gramática final de *C-mat* difere da apresentada nas fases anteriores do projeto, com destaque para a alteração no que diz respeito às regras gramaticais responsáveis por permitir a utilização do construto **if/else**. A seguir, encontra-se a gramática utilizada durante a apresentação final da disciplina a menos de construtos descartados, como **numListList**:

```
program : globalStmtList

globalStmtList : globalStmtList globalStmt
|

globalStmt : defFun
| declFun ;
| declOrdeclInitVar

declFun : ahead BASE_TYPE ID ( paramListVoid )

declOrdeclInitVar : typeAndNameSign ;
| typeAndNameSign = rvalue ;

typeAndNameSign : BASE_TYPE ID
| BASE_TYPE ID [ num ]
| mat BASE_TYPE ID [ num ] [ num ]

paramListVoid : paramList
|
paramList : paramList , param
| param

param : BASE_TYPE ID
| BASE_TYPE ID [ ]
| mat BASE_TYPE ID
localStmtList : localStmtList localStmt
|
localStmt : call ;
| lvalue = rvalue ;
| newFlowControl
| loop
| return expr ;
| IREAD ( lvalue ) ;
| FREAD ( lvalue ) ;
| print ( rvalue ) ;
| show ( rvalue ) ;
| print ( V_ASCII ) ;
| show ( V_ASCII ) ;

newFlowControl: countFlow flowControl

countFlow :

flowControl : if ( expr ) block else flowControl
| block
| ;

loop : while ( expr ) block
```

```
defFun : BASE_TYPE ID ( paramListVoid ) { declList localStmt }

numList : numList , num
| num

block : { localStmtList }

declList : declList declOrdeclInitVar
|

expr : expr ARITM expr
| expr % expr
| expr @ expr
| expr @@ expr
| expr REL_OP expr
| expr BIN_LOGI 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

num : V_INT
| V_FLOAT

lvalue : ID
| ID [ expr ]
| ID [ expr ] [ expr ]

rvalue : expr
| { numList }
```

Conforme pode ser notado, a gramática da linguagem se assemelha bastante à linguagem C. De fato, enquanto versão simplificada de C, era esperado alguma, ou ainda bastante similaridade em relação a ela. Entretanto, note-se algumas limitações ou

especificidades da gramática em relação a C, elencadas na subseção seguinte.

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). Caracteres que não são letras e estão nos corpos das regras são ou operadores ou delimitadores (parêntese, chave, vírgula, ponto e vírgula). O *token* ARITM pode ser qualquer um dos quatro operadores aritméticos básicos e REL_OP qualquer dos operadores relacionais existentes na linguagem C.

4.1 Limitações da gramática

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

1. declaração de variáveis apenas no escopo global ou no início de funções
2. matrizes são sempre bidimensionais
3. vetores são unidimensionais e suportam apenas os tipos **int** e **float**
4. declarações de variáveis locais devem ser realizadas todas antes de algum comando
5. necessidade da palavra reservada **ahead** ao se declarar previamente uma função sem corpo
6. necesidade de haver um *else* para cada *if*,
7. passagem de matriz como parâmetro e passagem de parâmetros em geral, a qual não precisa do operador **&** para passsar arrays e matrizes por referência. Contudo, escalares são sempre passados por cópia
8. ausência do operador **-** unário
9. ausência de operadores que manipulam bits (shift, operações lógicas bit-a-bit)

Tais limitações tiverem por meta a redução da complexidade do tradutor hoje implementado. Algumas não foram planejadas de início, como a necessidade de palavra chave exclusiva para declarar a assinatura de uma função antes de sua definição, ao passo que outras a exemplo do segundo item foram pensadas para reduzir o escopo do projeto e permitir maios probabilidade de conclusão no prazo estipulado pela professora.

Contudo, houve também cortes os quais não se mostraram necessários ao final do projeto, por conta da facilidade de implementação dessas operações em virtude da linguagem alvo desse tradutor: é o caso das últimas duas limitações elencadas acima, as quais ocorrem pois ao longo da implementação do projeto foi gasto bastante tempo à procura e resolvendo *bugs* e optou-se por não implementar tais funcionalidades. Entretando, posteriormente virificou-se a facilidade provida pela linguagem alvo no tocante a implementar tais funcionalidades.

4.2 Bugs na gramática

Para a fase de geração de código, a gramática em si não precisava ser tão alterada, mas implementar o *esquema de tradução* acabou por requerer algumas modificações, em especial a alteração da do corpo da regra *flowControl*, a qual continua sendo encaeável

consigo mesma mas agora pode também derivar um bloco ou ponto e vírgula. Contudo, dada a similaridade das ações semânticas que devem ser executadas caso qualquer operador aritmético (+, -, *, /) seja encontrado, o autor optou por remover as 4 regras e substituir por apenas uma nova. Tal estratégia não causa erro sintático nem impossibilita a implementação do analisador sintático.

Contudo, tal abordagem não funciona pois as precedências dos operadores aritméticos não são as mesmas entre todos. O correto seria a criação de uma função e simplesmente invocá-la durante a realização de ações semânticas em qualquer uma das regras de produção de expressões que contém tais operadores: expr + expr, expr - expr, expr * expr e expr / exp. Situação análoga ocorreu com os operadores lógicos **&&** e **||**, e também aos operadores relacionais >, <, >=, <=, != e ==. Entretanto, por estes últimos possuírem todos a mesma precência, não houve efeito colateral indesejado.

5 Semântica

Variáveis podem ser declaradas e inicializadas em apenas um comando, porém apenas uma pode ser declarada [e inicializa] por comando. Decisão tomada a fim de facilitar a implementação. Contudo, aqui há bugs relevantes, os quais estão devidamente reportados em 5.1.

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 (desde que o tipo destino seja superior ao tipo alvo), 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.

Os tipos de dados indexáveis em C-mat são matrizes e vetores, sendo as primeiras indexadas de maneira idêntica a vetores bidimensionais em C. Tais tipos são passados por referência para funções, enquanto inteiros e pontos flutuantes o são por cópia. Não é realizado nenhuma tipo de conversão entre tipos quando se invoca uma função; ou seja, o usuário deve garantir que os tipos dos argumentos passados batem com ordem e tipos dos parâmetros esperados pela função. Caso não haja tipos diferentes nos parâmetros e argumentos correspondentes, o compilador emite mensagem de erro.

Operadores binários deveriam ser 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:

1. adição/subtração
2. multiplicação/divisão
3. resto da divisão

4. comparação (maior que, menor que, maior ou igual a, menor ou igual a, igual a e não igual a)
5. AND e OR lógicos
6. NOT lógico
7. operador endereço (&)
8. multiplicação de matrizes
9. exponenciação de matrizes

Os níveis de precedência foram acidentalmente trocadas por conta da situação explicada na seção 4.2. Ao final, todos os operadores referidos nessa seção ficaram sendo associativos à esquerda, porém com o mesmo nível de precedência. Portanto, até que esse incômodo seja resolvido (ou seja, na próxima *release* de C-mat), recomenda-se a utilização de parênteses em basicamente qualquer expressão, por mais simples que seja. Resssalte-se que esse *bug* é de fácil correção e será corigido tão cedo quanto possível.

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 caso necessário (*narrow cast* deve ser feito explicitamente). 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 gerar 0 caso o operando não o seja e 1 caso contrário. Não é feita nenhuma checagem de tipos aqui; portanto, negar logicamente qualquer vetor ou matriz deve resultar em 0, visto que será negado o inteiro relativo ao endereço simbólico da base da estrutura composta.

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 (**int** e **float**). Matrizes podem ser contruídas enquanto tipos nativos da linguagem, em notação similar ao que seria um vetor bidimensional, porém com o adicional de ser necessário informar qual o tipo base da matriz 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.

5.1 Bugs

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 possuia 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

Primeiramenre, ressalte-se que boa parte da análise semântica foi realizada com sucesso; contudo, há algumas 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 contrução da AST
- *void* print_reshi: 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.

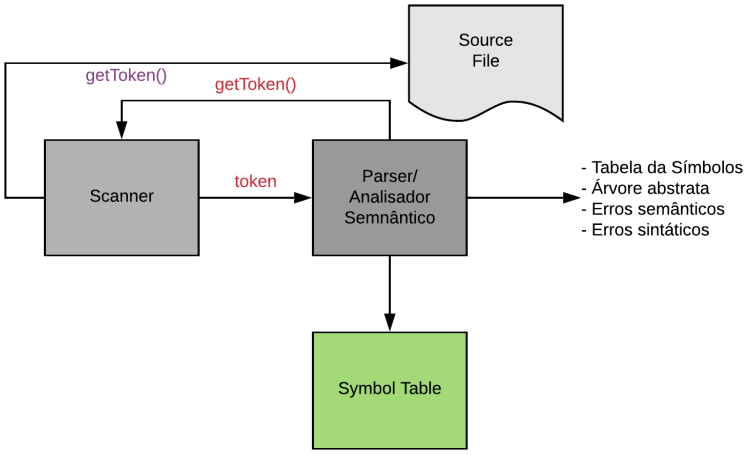
6.3 Funcionamento do programa

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

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 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.

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



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, prosegue-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 refeito. 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 contrução 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 contêm 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 13.

PROJECT, T. F. 2019. Acessado: 18-11-2019. Disponível em: <https://westes.github.io/flex/manual/>. Citado na página 13.

THEBISONTEAM. 2019. Acessado: 18-11-2019. Disponível em: <https://www.gnu.org/software/bison/manual/>. Citado na página 13.