ITGRAPHS: uma linguagem para percorrimento em topologias de grafos

Nur Corezzi - 15/0143290

Departamento de Ciência da Computação, Campus Darcy Ribeiro, Asa Norte, Brasília - DF, 70910-900

1 Introdução

Neste trabalho será apresentada a especificação de uma linguagem a ser desenvolvida e que se destina a área de processamento de grafos. A mesma tem como objetivo fornecer funcionalidades básicas de uma linguagem imperativa como comandos iterativos *for* e *while*, condicionais *if*, tratamento de expressões *booleanas* e aritméticas e também algumas funcionalidades orientadas a grafos. Seu diferencial está na introdução de um tipo básico *graph* que possui uma interface para inserção de arestas direcionais e armazenamento de valores em vértices. Também existirá um comando *for* facilitador que permite iterar fazendo percorrimentos *DFS* (Depth First Search) e *BFS* (Breadth First Search) buscando por valores específicos armazenados em cada vértice. A nova primitiva facilita buscas que necessitam ser feitas com frequência em grafos e permite que ações sejam seletivamente executadas de acordo com o percorrimento e os valores dos vértices armazenados. Este tipo básico poupa a necessidade do programador em manter e cuidar do uso de sua própria estrutura, permitindo que algum comportamento seja aplicado de forma prática à topologia de um grafo especificado.

2 Visão geral do tradutor

O tradutor implementado consiste em uma separação de processamento comumente utilizada no desenvolvimento de compiladores para linguagens de programação. As etapas consistem em *análise léxica, sintática, semântica* e por fim *geração de código intermediário*. Inicialmente será definida uma gramática para a linguagem, em seguida serão determinadas as unidades léxicas para gerar as definições do analisador léxico (gerado em FLEX) que será responsável por fornecer os *tokens* reconhecidos. O analisador sintático criado por *BISON* irá fazer uso dos *tokens* formatados para gerar a árvore de derivação e popular a tabela de símbolos. Por último o analisador semântico irá verificar erros e gerar as anotações devidas na árvore obtida para que em seguida seja feita a geração de código intermediário. Vale ressaltar que apesar da divisão de tarefas a tradução será realizada em apenas uma passada pelo código fonte.

3 Gramática

3.1 Definição

⟨*init*⟩ ::= ⟨*program*⟩

⟨*program*⟩ ::= ϵ
| ⟨*program*⟩ ⟨*function*⟩

⟨*function*⟩ ::= ⟨*type*⟩ ⟨*dimension*⟩ ⟨*id*⟩ '(' ⟨*params*⟩ ')' ⟨*block*⟩

⟨*params*⟩ ::= ϵ
| ⟨*params*⟩ ',' ⟨*declaration*⟩
| ⟨*declaration*⟩

⟨*function-call*⟩ ::= ⟨*id*⟩ '(' ⟨*params-call*⟩ ')'

⟨*params-call*⟩ ::= ϵ
| ⟨*params-call*⟩ ',' ⟨*expr-assign*⟩
| ⟨*expr-assign*⟩

⟨*graph-call*⟩ ::= 'dfs' ⟨*graph-params-call*⟩
| 'bfs' ⟨*graph-params-call*⟩

⟨*graph-params-call*⟩ ::= '(' ⟨*id*⟩ ',' ⟨*expr-assign*⟩ ',' ⟨*expr-assign*⟩ ')'

⟨*statements*⟩ ::= ϵ
| ⟨*statements*⟩ ⟨*statement*⟩

⟨*block*⟩ ::= '{' ⟨*statements*⟩ '}'

⟨*statement-control*⟩ ::= ⟨*block*⟩
| 'if' '(' ⟨*expr-assign*⟩ ')' ⟨*block*⟩
| 'if' '(' ⟨*expr-assign*⟩ ')' ⟨*block*⟩ 'else' ⟨*statement-control*⟩

⟨*statement*⟩ ::= ⟨*statement-control*⟩
| 'while' '(' ⟨*expr-assign*⟩ ')' ⟨*block*⟩
| 'for' '(' ⟨*expr-assign*⟩ ';' ⟨*expr-assign*⟩ ';' ⟨*expr-assign*⟩ ')' ⟨*block*⟩
| 'for' '(' ⟨*id-or-access*⟩ ':' ⟨*graph-call*⟩ ')' ⟨*block*⟩
| '>>' ⟨*id-or-access*⟩ ';'
| '<<' ⟨*expr-assign*⟩ ';'
| ⟨*declaration*⟩ ';'
| ⟨*expr-assign*⟩ ';'
| ⟨*graph-add*⟩ ';'
| 'return' ';'
| 'return' ⟨*expr-assign*⟩ ';'

⟨*expr-assign*⟩ ::= ⟨*expr-relational*⟩ '=' ⟨*expr-assign*⟩
| ⟨*expr-or*⟩

⟨*expr-and*⟩ ::= ⟨*expr-or*⟩ ⟨*and*⟩ ⟨*expr-and*⟩
| ⟨*expr-or*⟩

⟨*expr-or*⟩ ::= ⟨*expr-relational*⟩ ⟨*or*⟩ ⟨*expr-or*⟩
| ⟨*expr-relational*⟩

⟨*expr-relational*⟩ ::= ⟨*expr-add*⟩ ⟨*compare-op*⟩ ⟨*expr-relational*⟩
| ⟨*expr-add*⟩

⟨*expr-add*⟩ ::= ⟨*expr-sub*⟩ ⟨*add*⟩ ⟨*expr-add*⟩
| ⟨*expr-sub*⟩

⟨*expr-sub*⟩ ::= ⟨*expr-mul*⟩ ⟨*sub*⟩ ⟨*expr-sub*⟩
| ⟨*expr-mul*⟩

⟨*expr-mul*⟩ ::= ⟨*expr-div*⟩ ⟨*mul*⟩ ⟨*expr-mul*⟩
| ⟨*expr-div*⟩

⟨*expr-div*⟩ ::= ⟨*expr-unary*⟩ ⟨*div*⟩ ⟨*expr-div*⟩
| ⟨*expr-unary*⟩

⟨*expr-unary*⟩ ::= ⟨*unary*⟩ ⟨*factor*⟩
| ⟨*factor*⟩

⟨*factor*⟩ ::= '(' ⟨*expr-assign*⟩ ')'
| ⟨*value*⟩
| ⟨*function-call*⟩

⟨*unary*⟩ ::= '!'
| ⟨*add*⟩
| ⟨*sub*⟩

⟨*and*⟩ ::= '&&'

⟨*or*⟩ ::= '||'

⟨*add*⟩ ::= '+'

⟨*sub*⟩ ::= '-'

⟨*mul*⟩ ::= '*'

⟨*div*⟩ ::= '/'

⟨*compare-op*⟩ ::= '<' | '<=' |'>' |'>=' | '==' | '!='

⟨*graph-add*⟩ ::= 'addv' '(' ⟨*id-or-access*⟩ ')'
| 'adda' '(' ⟨*id-or-access*⟩ ',' ⟨*expr-assign*⟩ ',' ⟨*expr-assign*⟩ ')'

⟨*declaration*⟩ ::= ⟨*type*⟩ ⟨*dimension*⟩ ⟨*id*⟩

⟨*dimension*⟩ ::= ϵ
| '[' ⟨*number-int*⟩ ']'

⟨*value*⟩ ::= ⟨*id-or-access*⟩
| ⟨*number*⟩
| ⟨*boolean-const*⟩

⟨*id-or-access*⟩ ::= ⟨*id*⟩
| ⟨*id*⟩ ⟨*access-lvl*⟩

⟨*access-lvl*⟩ ::= ϵ
| '[' ⟨*expr-assign*⟩ ']' ⟨*access-lvl*⟩

⟨*type*⟩ ::= 'int'
| 'boolean'
| 'float'
| 'graph'
| 'void'

⟨*number*⟩ ::= ⟨*number-int*⟩
| ⟨*number-float*⟩

⟨*number-int*⟩ ::= ⟨*digit*⟩ ⟨*number-list*⟩

⟨*number-float*⟩ ::= ⟨*digit*⟩ ⟨*number-list*⟩ '.' ⟨*digit*⟩ ⟨*number-list*⟩

⟨*number-list*⟩ ::= ⟨*number-list*⟩ ⟨*digit*⟩
| ϵ

⟨*digit*⟩ ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

⟨*boolean-const*⟩ ::= 'true'
| 'false'

⟨*id*⟩ ::= ⟨*letter*⟩ ⟨*letter-or-digit-list*⟩

⟨*letter-or-digit-list*⟩ ::= ⟨*letter-or-digit-list*⟩ ⟨*letter*⟩
| ⟨*letter-or-digit-list*⟩ ⟨*digit*⟩
| ' '

⟨*letter*⟩ ::= 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z' | 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z' | '_'

1. Refatoração de *statements* comandos de *if-else while* e *for* foram simplificados para sempre requisitarem corpo com abertura e fechamento de aspas. A estrutura anterior estava dificultando demasiadamente a geração de código intermediário.
2. Adição de *return* sem parâmetro.
3. Mudança para que expressões relacionais sejam avaliadas antes de *or* e *and*

3.2 Aspectos informais da gramática

A gramática em questão define uma lista de funções como um programa válido, tendo ponto de execução inicial em uma função definida com identificador *main*. Cada função possui um conjunto de argumentos que são declarações que envolvem um tipo e um identificador e possui também um tipo de retorno. Serão também delimitadas por um conjunto de abre e fecha chaves {} que possui uma lista de declarações.

Uma declaração constitui um comando da linguagem que podem ser qualquer um dos especificados em *statements*. Existe também um comando diferenciado de *for* para o percorrimento dos grafos que contempla também chamadas de funções *BFS* e *DFS* na regra *graph-op*. Esta chamada recebe um identificador do grafo um parâmetro vértice de início e um valor a ser procurado, a função irá retornar os vértices na ordem em que o valor especificado for encontrado no percorrimento (exemplo em código mais abaixo).

Um grafo irá iniciar com zero vértices e possui duas funções auxiliares para acrescentar vértices(*addv*) e arestas(*adda*) de forma dinâmica. Cada vértice armazena um valor inteiro que pode ser acessado por colchetes e será sempre iniciado em zero. Esta decisão evita tipos genéricos na definição apenas para simplificar a implementação. Com relação as operações (descritas mais abaixo em código) será possível atribuir valores a um vértice, identificado por um inteiro, e designar arestas entre vértices existentes.

Os comandos de *for*, *while* e *if* contemplam expressões que irão avaliar os fluxos de execução e também uma regra *statement* que permite encadear uma lista de comandos a sua execução.

Expressões podem ser consideradas como operações aritméticas, lógicas e atribuições, que seguem uma precedência nessa mesma ordem. Existe também a possibilidade de definir prioridades distintas a partir do uso de parênteses definidos na regra *factor*.

Os tipos básicos da linguagem contemplam *boolean*, *int*, *float*, *void* e *graph*. O tipo graph é um tipo especial que também é unicamente identificado pelo tamanho especificado entre colchetes *[number-int]*. Todos os tipos podem também ser declarados como *array* unidimensional (indexados a partir de zero) com exceção de *void* que apenas pode estar presente em declaração de funções.

Por último os numerais da linguagem são definidos pelos terminais numéricos como listas que definem inteiros e reais e os identificadores são definidos pelos carácteres alfabéticos minúsculos e maiúsculos além do underscore.

3.3 Descrição semântica

A seguir segue um pequeno trecho de código explicitando o uso da nova primitiva em um programa básico da linguagem:

```
void main() {
    graph g;
    // Acrescenta um vrtice
```

```
    addv(g);
    //Define uma aresta direcional
    adda(g,0,2)
    // Designa valor para algum vertice
    g[0] = g[2] = 1;
    int vertice;
    for (vertice : dfs(g, 0, 1)) {
        // Percorre o grafo em dfs iniciando no vertice zero, procurando
        //  vrtices  que possuam o valor 1 armazenado
        << vertice;
    }
}
```

O ponto inicial de execução será sempre a função que possui o identificador de *main* e sua falta implica em não execução e será gerado um *warning*. As funções devem ser declaradas antes de serem utilizadas assim como as variáveis. Parâmetros de tipos básicos *boolean*, *int* e *float* serão passados por cópia porém *arrays* e *graph* serão passados por referência. Apenas será possível passar um *array* se tanto o destino como a fonte tiverem o mesmo tipo, o que é estabelecido pelo tamanho, dimensões e tipo armazenado. Não será possível retornar *array* nem *graph* de funções, apenas os tipos *boolean*, *int*, *float* e *void*.

Arrays terão seu tamanho estabelecido em tempo de compilação e apontarão para a mesma região de memória ao longo da execução do programa não podendo ser realocados ou apontar para outro *array*. O tipo *graph* também irá se comportar de forma análoga sem poder participar de atribuições pelo usuário porém será uma estrutura dinâmica em que suas dimensões podem crescer.

Com relação a *casts* implicitos o único necessário e existente será entre *float* e *int*. O tipo *boolean* se trata de um *int* onde qualquer valor diferente de zero será *true* e caso contrário false. Caso exista um *float* na operação o *cast* dos operandos será para *float*, e caso destino seja *int* ou *boolean* será efetuado um *cast* de *float* para *int*. Portanto na árvore abstrata apenas existiram anotações de conversões *floatToInt* e *intToFloat*. Vale ressaltar também que expressões de atribuição possuem um valor de retorno que será o do próprio identificador sendo atribuído

O escopo das variáveis é delimitado pelas chaves. A busca de uma variável na tabela se resume a achar a ocorrência do símbolo desejado onde o escopo atual está contido no escopo do símbolo encontrado. Caso hajam diversos símbolos na hierarquia que sejam candidatos então o símbolo do escopo mais próximo acessível será escolhido. Antes de ser inserida uma variável a mesma busca é efetuada para verificar redeclarações. A busca também se aplica para funções com uma restrição adicional de que as mesmas são identificadas pelo seu nome de declaração portanto não podem existir funções distintas com o mesmo nome, mesmo que a lista de parâmetros seja diferente.

Os comandos condicionais e iterativos exigem que a declaração de variáveis sejam feitas fora de sua assinatura e possuem um escopo próprio (com exceção de *if* simples sem chaves). O iterador de grafos *for* exige que o parâmetro recebendo

o vértice de retorno seja um *int* uma vez que os vértices apenas são identificados como inteiros.

4 Implementação

Para a realização da análise léxica foi utilizado o gerador de analisador léxico *FLEX*. Foi realizada a definição regular para reconhecimento dos lexemas para que em seguida fosse transportada para as definições em *FLEX*. Os tokens reconhecidos são repassados como um enumerador, a informação adicional sobre o lexema é encontrada nas próprias variáveis de *buffer* que o gerador produz podendo ser acessadas pelo analisador sintático com facilidade.

O gerador de analisador sintático escolhido foi o BISON. Estes gerador cria analisadores *top-down* LALR(1) e facilita o procedimento de criação dos conjuntos e estados LR, gerando também as tabelas de ações e os procedimentos necessários para a derivação da árvore sintática. Seguindo a especificação da documentação, a gramática formalmente apresentada acima foi traduzida para o formato de produções compreendidas pelo BISON. As definições de tokens e regras da gramática podem ser encontradas no arquivo font *gramatica.y*.

Para cada regra da gramática foram associadas ações semânticas para a criação da árvore abstrata e análise semântica. Cada nó da árvore é representado por uma estrutura genérica *Node* que irá conter um identificador do símbolo da gramática o qual representa para auxiliar em ações posteriores. Foram abstraídas rotinas adicionais para criação destes nós, assim como para a criação e manipulação da tabela de símbolos que serão melhor especificadas abaixo.

4.1 Definição regular

Na Figura 1 é possível identificar as variáveis em negrito/itálico e os símbolos terminais em fonte normal. Não foram agrupados símbolos por funcionalidade como comparadores, operadores, tipos e sim um token por unidade terminal da linguagem. Caso seja necessário agrupamentos para facilitar a análise sintática, posteriormente podem ser definidas funções que determinam pertinência de determinado token em algum grupo. Cada token será identificado por um valor de *enum* gerado pelo analisador sintático. Os tokens são especificados em *gramatica.y* pelas regras prefixadas em *%token*.

4.2 Estrutura da árvore sintática abstrata (***AST***)

Á árvore gerada é construída por meio de uma abstração de um nó. Cada nó pode conter diversos filhos e possuem um identificador para informar qual símbolo da gramática o gerou. Os nós são criados individualmente a cada redução e a estrutura da árvore é gerada a partir do método de adição de filhos. Cada regra composta por mais de um símbolo, que já possua valor semântico presente (ou seja nó instanciado), será acrescida como filho do símbolo ao qual a regra será reduzida. Derivações que geram listas terão seu formato em árvore simplificado para

ws	→ [\t\n]	***read***	→ >>
letter	→ [A-Za-z]	***write***	→ <<
digit	→ [0-9]	***and***	→ &&
		or	→ ||
if	→ if	***le***	→ <=
else	→ else	***ge***	→ >=
for	→ for	***less***	→ <
while	→ while	***greater***	→ >
		eq	→ ==
boolean	→ boolean	***ne***	→ !=
int	→ int	***not***	→ !
float	→ float	***mul***	→ *
void	→ void	***div/***	→ /
graph	→ graph	***sum***	→ +
to	→ to	***sub***	→ -
		assign	→ =
true	→ true	***end***	→ ;
false	→ false	***open_brace***	→ {
		close_brace	→ }
dfs	→ dfs	***it***	→ :
bfs	→ bfs	***separator***	→ ,
return	→ return	***open_p***	→ (
		close_p	→)
id	→ (***letter***) (***letter*** | ***digit*** | _)*	***open_bracket***	→ [
number	→ (***digit***)+ (. ***digit***+)?	***close_bracket***	→]

Figura 1. Definições regulares, number foi separado em c_int e c_float para facilitar identificação.

listas ligadas, o que facilita alguns processamentos. Também houve a remoção de nós não essenciais da sintaxe após as reduções, como vírgulas, parênteses e etc. Ao final, na regra geradora o valor será atribuído á um ponteiro AST declarado no analisador.

Cada nó possui uma informação de qual símbolo da gramática o mesmo representa como por exemplo uma função, parâmetro, tipo, identificador e etc. Os nós também possuem ponteiros para seus filhos e além disso uma informação adicional de complemento utilizada por variáveis, funções e constantes numéricas que precisam do valor ao qual se referem. Nós também possuem uma *TypeExpression* que define seu tipo na gramática. O tipo será utilizado para verificações de operações possíveis e conversões em *casts* ímplicitos. As conversões também serão anotadas na árvore especificamente no nó com um campo informando se a conversão deve ser *floatToInt* ou *intToFloat*, que por enquanto são as duas únicas conversões possíveis. Vale ressaltar que por simplicidade os tipos *boolean* são traduzidos e tratados como tipos *int* pelo tradutor.

4.3 Tabela de símbolos

A tabela de símbolos inicialmente é uma estrutura de lista de listas ligada onde cada colisão gera um novo encadeamento em alguma entrada. Os elementos instanciados são declarações de variáveis e de funções. As informações adicionadas se tratam de um enumerador indicando se é uma função ou variável, um identificador com o nome da variável ou função, o tipo do símbolo na gramática (*float*, *int*, *graph<int>*, *arrays* e etc), que irá facilitar verificações e uma referência para o nó da árvore abstrata caso seja necessário ter acesso a informações adicionais, como por exemplo os parâmetros de funções para verificações semânticas.

Também foi adicionada uma informação adicional de escopo. Cada entrada da tabela possui um escopo próprio que é construído juntamente com a árvore abstrata. A cada abertura de chaves um novo escopo é adicionado na pilha de escopo global e são representados por um inteiro obtido de um contador. O escopo corrente em qualquer momento será a concatenação de todos os escopos nesta pilha e será também a informação adicionada juntamente com os símbolos na tabela. Esta informação será utilizada na busca de símbolos na tabela para que um símbolo seja referenciado de forma que faça sentido seguindo a estrutura da árvore abstrata construída.

4.4 Gerador de código intermediário

A tradução para *Three Adress Code*(TAC) será realizada em uma passagem, portanto será necessário a realização de *backpatching* para a resolução de pulos para rótulos ainda não definidos. Cada nó possui uma lista *next* que irá conter os campos que devem ser atualizados com algum rótulo futuro. Expressões serão avaliadas por completo evitando a realização de *short-circuit* para simplificação das rotinas de geração de expressões. Cada expressão gerada na linguagem será atribuída a uma variável temporária em TAC e poderá ser acessada a partir do código que irá ser armazenado em cada nó da árvore abstrata.

Cada inicio de função reinicia o contador de variáveis temporárias para que possam ser reutilizadas no novo contexto. Devido a falta de alocações estáticas globais, todas as declarações na linguagem irão gerar um endereço armazenado em variável temporária e a sua referência será armazenada na tabela de símbolos para posteriores acessos e liberação de alocações. Já para parâmetros de funções seus valores serão armazenados em pilha e o símbolo na tabela não será um temporário e sim uma referência para a variável em pilha.

Para *arrays* serão armazenados no inicio do endereço do código em TAC o tamanho fazendo com que exista um *offset* para os reais valores armazenados. Endereços em TAC sempre serão acompanhados de um índice para seu acesso, uma vez que operações aritméticas com endereços não devem ser realizadas.

Variáveis do tipo *graph* irão alocar inicialmente três espaços de memória, um caracterizando o tamanho do grafo, outro ponteiro para o *array* de valores dos vértices e um terceiro ponteiro para as arestas do grafo. A lista de arestas é um *array* de ponteiros para listas uma vez que cada lista de arestas pode conter um tamanho variado e portanto não serão armazenados de forma contígua. Cada adição de aresta ou vértice gera novas alocações e atualizam os valores nos campos armazenados na tabela de símbolos.

Para facilitar os procedimentos de geração de código foi gerada uma estrutura que define instruções de código em TAC assim como funções para manipular a geração. Cada *Instruction* possui um rótulo, o comando em TAC o qual representa e os parâmetros representados por três campos *Field*. Instruções que não utilizam todos os campos ou rótulos apenas devem passar valores nulos nos campos não desejados. Também existe um campo *next* em instruções que corresponde a próxima instrução encadeada. Funções auxiliares foram geradas para facilitar a criação de campos, rótulos, instruções, junção de instruções e geração de estruturas mais complexas na linguagem como por exemplo declarações ou rotinas de *output* e *input* as quais serão descritas mais abaixo.

4.5 Descrição de funções adicionais

Funções adicionais foram criadas para: auxiliar na criação da tabela de símbolos, gerar os nós da árvore abstrata, gerar código intermediário, fazer liberação de alocações e auxiliar na impressão das estruturas para o console. A seguir se encontram algumas das funções adicionais utilizadas pelo programa tradutor:

Árvore sintática abstrata

1. *create_node* e *push_child*: Responsáveis por criar um nó *standalone* e adicionar filhos a algum nó.
2. *free_node*, *free_tree*: Responsáveis por liberação de memória alocada para a árvore abstrata.
3. *print_node_prefix*, *print_node_sufix*, *print_node* e *print_tree*: Funções auxiliares para imprimir no console a árvore abstrata formatando os prefixos e separando nós de forma individual para debug.

Tabela de símbolos

1. *stable_create_symbol*, *stable_find* e *stable_add*: Funções auxiliares para criar um símbolo *standalone*, identificar a existência de um símbolo na tabela e adicionar um símbolo qualquer a tabela.
2. *stable_find_with_scope*: Realiza uma busca na tabela levando em consideração um escopo atual. Será utilizada em conjunto com um ranqueamento de acesso a escopos para determinar símbolo mais próximo do escopo atual.
3. *free_symbol_node* e *free_stable*: Responsáveis por liberação de memória alocada para a tabela de símbolos.
4. *stable_symbol_print* e *stable_print*: São funções auxiliares para imprimir no console símbolos individuais e tabelas de símbolos.

Escopo

1. *scope_access_rank*: Estabelece um rank de acessibilidade de um escopo por outro, irá informar se um escopo B pode acessar um escopo A e se é acessível qual o nível de diferença entre os mesmos. Função irá auxiliar nas buscar da tabela de símbolos.
2. *scope_push* e *scope_pop*: Irá atualizar uma referência global de escopo em formato de pilha, *scope_push* em particular recebe uma referência da função atual na tabela de símbolos para facilitar alguns processamentos.
3. *scope_sz*, *scope_eq*, *scope_create*, *cpy_scope*, *print_scope* entre outras: São funções auto-explicativas que fornecem auxilio no tratamento de escopos.

Expressões de Tipos

1. *type_get_cast*: Retorna o tipo de cast que será necessário entre um tipo de origem e um tipo de destino.
2. *type_max*: Em uma hierarquia de tipos retorna o tipo comum mais próximo entre os dois tipos passados como parâmetros.
3. *type_from_access_lvl*: Devido a acesso de variáveis por colchetes é necessário inferir o tipo sendo tratado no acesso.
4. *type_can_assign*, *type_can_return*, *type_is_aritmetic* e *type_is_boolean*: Informa como alguns tipos podem ser tratados, se podem estar em uma atribuição, retorno de função, em uma expressão aritmética ou expressão booleana.
5. *type_build*, *type_cpy*, *type_eq*, *free_type*, *type_print* entre outras: São funções também auto-explicativas e fornecem abstrações para as funções acima e funcionalidades gerais ao tradutor.

Gerador de código intermediário

1. *cgen_label*, *cgen_field* e *cgen_instruction*: Funções auxiliares para gerar as instruções em TAC.

2. *cgen_patch*, *cgen_cpy_patch*, *cgen_merge_patch* e *cgen_back_patch*: Auxiliam no tratamento e resolução de *labels*.
3. *cgen_declaration* e *cgen_declaration_param*: geram código de declaração de variáveis em parâmetros e dentro de funções, também populam a tabela de símbolos com referência para os campos que guardam os endereços alocados.
4. *cgen_function_call* e *cgen_declaration_param*: Geram código para declaração de variaveis em parametro de função e de chamadas de função.
5. *cgen_var_access*: Gera código para acesso a valor de variáveis, resolve casos em que o que será retornado será um *lvalue* ou um *rvalue*.
6. *cgen_expression_relational*, *cgen_expression_boolean* e *cgen_expression_aritmetic*: Geram código para as expressões da linguagem.
7. *cgen_if* e *cgen_if_else*: Geram código para as estruturas de controle.
8. *cgen_write* e *cgen_write_array*: Gera instruções de escrita em console para variáveis com *lvalue* e *rvalue*.
9. *print_instruction* e *print_code*: Geram código TAC em saída padrão.

5 Tratamento de erros

5.1 Identificação e tratamento de erros léxicos

A nível de análise léxica apenas será necessário verificar erros referentes a caracteres não existentes e padrões que contemplem símbolos do alfabeto da linguagem mas que não se encaixem nas definições regulares informadas. A forma de resposta será sempre informar caractere a caractere do padrão não reconhecido informando linha e coluna. Em seguida todos os caracteres incorretos serão descartados da análise e o analisador irá continuar com o restante da entrada. Não foi necessário a introdução de procedimentos para tratamento de erro uma vez que a regra de captura de padrões não reconhecidos (regra ". {}"em FLEX) irá automaticamente descartar os caracteres não desejados bastando apenas acrescentar uma rotina para informar a ocorrência do erro.

5.2 Identificação e tratamento de erros sintáticos

Por simplicidade nenhum erro identificado será corrigido. O *token* responsável pelo erro será simplesmente descartado e o analisador por padrão irá desempilhar os estados até encontrar alguma derivação que possua um *token error*. Este *token* se apresenta em pontos chaves do código sendo eles a regra de produção de um função e a regra de produção de *statements* para que a partir destes pontos o analisador possa retomar apenas para verificar possíveis próximos erros na derivação. Os erros serão descritos identificando a linha e coluna do *token* o qual gerou o erro, e será informado qual *token* deveria ser esperado, sempre que possível inferir.

5.3 Identificação e tratamento de erros semânticos

Um conjunto não exaustivo de erros semânticos foi determinado levando em consideração o tempo disponível para implementação. Os erros são informados com a mensagem respectiva a linha e coluna do lexema que pode ter gerado o erro. A seguir seguem os erros:

1. Variável ou função já declarada, caso já apresentem uma ocorrência de mesmo escopo na tabela de símbolos.
2. Variável ou função não encontrada no escopo, caso não exista ocorrência na tabela de símbolos.
3. Operandos incompatíveis em expressões aritméticas e boleanas como por exemplo operações entre *arrays* e grafos que não existem na linguagem.
4. Tipos a esquerda em expressões de atribuição apenas podem ser *lvalue*.
5. Chamada indevida caso os tipos instanciados para a chamada de função não possuam uma equivalência de tipos.
6. Incompatibilidade do tipo em *return* e tipo da função.
7. Incompatibilidade de tipos em operações de atribuição.
8. *void* não pode ser tipo em varíavel nem array.
9. Funções não podem retornar *array* nem *graph*.
10. Tamanho da dimensão de *arrays* na declaração não pode ser zero.

6 Arquivos de teste

6.1 Arquivos de teste léxico

Os arquivos *lexico/correto1*, *lexico/correto2*, *lexico/errado1* e *lexico/errado2* possuem exemplos de programas sem e com erros léxicos. Em *errado1* são apresentados caracteres não reconhecidos pela linguagem, espera-se que sejam informados um a um os caracteres incorretos, e em *errado2* erros com símbolos do alfabeto da linguagem, porém com padrões não reconhecidos onde será apresentado um a um os caracteres que não fazem parte de um padrão reconhecível. Ambos arquivos de erro são descritos na Figura 4, lembrando que os comentários apenas descrevem o erro e não necessariamente o que será informado ao usuário.

```
void main() {
    @"'\çá~^  // Cada simbolo será informado como erro
}
```

```
void main() {
    .9 1.       // Numeros incorretos
    a | b       // Apenas um pipe não reconhecido
    a &&& b     // & a mais
    _nome       // Underscore no inicio de identificador
}
```

Figura 2. Programas comentados com os respectivos erros léxicos

6.2 Arquivos de teste sintáticos

Os arquivos *sintatico/correto1*, *sintatico/correto2*, *sintatico/errado1* e *sintatico/errado2* possuem exemplos de programas sem e com erros sintáticos. Em *errado1* é apresentada uma função sem tipo de retorno declarado, um for sem declaração de uma variável na primeira parte e uma declaração de variável sem fechamento de ponto e vírgula. Em *errado2* existem erros de comando de leitura sem variável de destino, for iterador sem uma função correta de iteração, um if sem condicional e um acesso incorreto de um vértice por meio de ponto flutuante. Em cada caso espera-se que o token imediato do erro seja informado e em algumas situações, onde seria possível inferir o token esperado, o mesmo também será informado. Em alguns casos é possível inferir o que seria o token esperado mas em muitas situações existe ambiguidade no que seria a intensão do usuário, portanto algumas mensagens não são precisas e difíceis de se determinar.

```
main() {            // Função sem tipo será identificado como ID não esperado
    int a, b, c;    // Todos as construções já reduzidas que compõem a função serão descartadas
}

void main () {
    int a = 2           // Seria esperado um ';' porém foi encontrado o token FOR
    for (; i; i) {}     // For sem declaração de índice para iterar, não é possível determinar um erro exato
                        // pois neste caso diferentes contruções corretas poderiam ser sugeridas
    while(a) a;
}
```

```
void main() {
    >>;                             // ';' não esperado seria esperado algum valor de uma expressão como ID
    for (int i : vdsf(a, 1, 2)) {   // FOR de iteração deve conter alguma função DFS ou BFS

    }
    if () {}                        // IF sem expressão para avaliar, final de parentesis não esperado
    g[1.2];                         // Acesso a vértice deve ser por inteiros, porém aqui mais de uma
                                    // construção será possível portanto mensagem talvez não seja significativa
}
```

Figura 3. Programas comentados com os respectivos erros sintáticos

6.3 Arquivos de teste semânticos

Os arquivos *semantico/correto1*, *semantico/correto2*, *semantico/errado1* e *semantico/errado2* possuem exemplos de programas sem e com erros semânticos. Os erros esperados foram descritos nas figuras abaixo com comentários ao lado da linha em que se espera que o erro ocorra.

7 Dificuldades da implementação

Com relação a parte léxica as definições da linguagem se mostraram simples e não exigiram grande aprofundamento no ferramental fornecido pelo *FLEX*. Na parte sintática já foi necessário maior estudo com relação as funcionalidades fornecidas

```
void main() { }

void main () {      // Função redeclarada
    int a;
    a = 2;
    int a;          // Váriavel redeclarada
    {
        int a;
        outra();    // Função não encontrada no escopo
        b = 2;      // Variável não encontrada no escopo/tipos incompatíveis
        graph c;
        c = a;      // Tipos incompativeis em atribuição
        1 = 1;      // Variável a esquerda em atribuição de ser lvalue
    }
    int b;
}
```

```
graph outra(int a) {    // graph não pode ser retorno
    return a;           // Tipo incompativel com retorno
}

void main() {
    graph g;
    int a;
    a = 1 + g;          // Tipos incompativeis em expressão e atribuição
    outra(g);           // Chamada com parâmetros incompativeis
    outra();            // Chamada com parâmetros incompativeis
}
```

Figura 4. Programas comentados com os respectivos erros sintáticos

pelo *BISON*. Ambiguidades presentes na gramática também apresentaram desafios na sua redefinição. Também existiram dificuldades com relação ao que seria necessário para a próxima fase uma vez que não havia muito conhecimento sobre o que seria feito na análise semântica nem na geração de código intermediário. Essa dificuldade já se apresentou na parte léxica uma vez que mudanças foram efetuadas nesta fase devido ao desconhecimento de certas necessidades.

Na parte semântica a maior dificuldade foi determinar um sub-conjunto de erros e conversões que fossem possíveis ser tratados no tempo disponível para implementação. A verificação da existência de um retorno em funções foi descartada pois exige a verificação de todos os caminhos possíveis no código de uma função o que é dificultado pelos comandos condicionais. Também foram descartados *cast* entre *arrays* uma vez que podem se apresentar de forma complexa na geração de código intermediário. Também foram realizadas algumas simplificações na linguagem para poder generalizar melhor os tratamentos sendo efetuados nesta fase.

Referências

[ALSU07] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2 edition, jan 2007.

[Dega] Jutta Degener. ANSI C grammar, lex specification. http://www.quut.com/c/ANSI-C-grammar-l-2011.html. Accessed: 2020-03-17.

[Degb] Jutta Degener. ANSI C yacc grammar. http://www.quut.com/c/ANSI-C-grammar-y-2011.html. Accessed: 2020-03-17.