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

Foi levantado que a implementação do grafo como matriz ficaria muito simplificada e sugerido a implementação de grafos dinâmicos. O documento ainda não foi atualizado por falta de tempo, mas pretende-se implementar a sugestão liberando a adição de vértices e possivelmente adicionando a criação de um tipo genérico para que o armazenamento do valor do vértice não seja somente tipos inteiros. Com relação a adição de rótulos para vértices será adicionado caso sobre tempo para a sua adição.

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 impeprativa 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 sintático (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*⟩ ⟨*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*⟩

⟨*statement*⟩ ::= ⟨*statement-prefix*⟩ ⟨*statement-end*⟩
| ⟨*statement-prefix*⟩ ⟨*dangling-if*⟩
| ⟨*statement-end*⟩
| ⟨*dangling-if*⟩

⟨*statement-no-dangle*⟩ ::= ⟨*statement-prefix*⟩ ⟨*statement-end*⟩
| ⟨*statement-end*⟩

⟨*dangling-if*⟩ ::= 'if' '(' ⟨*expr-assign*⟩ ')' ⟨*statement*⟩

⟨*statement-prefix*⟩ ::= 'if' '(' ⟨*expr-assign*⟩ ')' ⟨*statement-no-dangle*⟩ 'else'
| 'while' '(' ⟨*expr-assign*⟩ ')'
| 'for' '(' ⟨*declaration-or-assign*⟩ ';' ⟨*expr-assign*⟩ ';' ⟨*expr-assign*⟩ ')'
| 'for' '(' ⟨*declaration*⟩ ':' ⟨*graph-call*⟩ ')'

⟨*block*⟩ ::= '{' ⟨*statements*⟩ '}'

⟨*statement-end*⟩ ::= ⟨*block*⟩
| '>>' ⟨*id-or-access*⟩ ';'
| '<<' ⟨*expr-assign*⟩ ';'
| ⟨*declaration-or-assign*⟩ ';'
| ⟨*expr-assign*⟩ ';'
| ⟨*graph-add*⟩ ';'
| 'return' ⟨*expr-assign*⟩ ';'

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

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

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

⟨*expr-or*⟩ ::= ⟨*expr-add*⟩ ⟨*or*⟩ ⟨*expr-or*⟩
| ⟨*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*⟩ ::= ⟨*id*⟩ '[' ⟨*expr-assign*⟩ 'to' ⟨*expr-assign*⟩ ']'

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

⟨*declaration*⟩ ::= ⟨*type*⟩ ⟨*size-specifier*⟩ ⟨*id*⟩

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

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

⟨*id-or-access*⟩ ::= ⟨*id*⟩
| ⟨*id*⟩ '[' ⟨*number-int*⟩ ']'

⟨*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. Acrescentados *number-int* e *number-float* para facilitar identificação.
2. Estrutura de *statements* foi modificada devido a ambiguidades. Em *statement-prefix* é forçado o casamento entre um *if* e um *else*, impedindo *if* abertos dentro de uma sessão *if-else* a não ser que o mesmo esteja em uma sessão de bloco.
3. Adição das constantes *false* e *true* que possuem valor semântico 0 e 1 respectivamente.
4. Adição de *size-specifier* para especificação do tamanho do grafo na declaração.
5. Adição de operadores de comparação que haviam sido esquecidos.

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 terá o número de vértices especificado na sua declaração não sendo possível a remoção ou adição de mais vértices. Cada vértice armazena apenas valores inteiros. 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 parêntesis 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]*.

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. 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 grafos 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 tanto pelo tamanho quanto pelo tipo armazenado. Não será possível retornar *array* de funções, apenas os tipos *boolean*, *int*, *float* e *graph*.

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*. Já o tipo *graph* poderá receber uma nova referência e será sempre declarado inicialmente sem arestas e com valores de vértices padrão em zero.

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*.

1. Estrutura anterior de token foi removida poís se apresentou desnecessária. Agora o valor passado ao analisador sintático é apenas um inteiro representando o token, o valor do lexema é extraído de variáveis produziadas pelo BISON, assim como as linhas e colunas para indicar posição.

ws → [\t\n]
letter → [A-Za-z]
digit → [0-9]

if → if
else → else
for → for
while → while

boolean → boolean
int → int
float → float
void → void
graph → graph
to → to

true → true
false → false

dfs → dfs
bfs → bfs
return → return

id → (***letter***) (***letter*** | ***digit*** | _)*
number → (***digit***)+ (. ***digit***+)?

read → >>
write → <<
and → &&
or → ||
le → <=
ge → >=
less → <
greater → >
eq → ==
ne → !=
not → !
mul → *
div/ → /
sum → +
sub → -
assign → =
end → ;
open_brace → {
close_brace → }
it → :
separator → ,
open_p → (
close_p →)
open_bracket → [
close_bracket →]

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

4.2 Descrição de funções adicionais

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

1. *create_node* e *push_child*: Responsáveis por criar um nó *standalone* e adicionar filhos a algum nó.
2. *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.
3. *free_node*, *free_tree*, *free_symbol_node* e *free_stable*: responsáveis por liberação de memória alocada para a tabela de símbolos e para a árvore abstrata.
4. *stable_symbol_print* e *stable_print*: São funções auxiliares para imprimir no console símbolos individuais e tabelas de símbolos.
5. *print_node_prefix*, *print_node_sufix*, *print_node* e *print_tree*: Também funções auxiliares para imprimir no console a árvore abstrata formatando os prefixos e separando nós de forma individual para debug.
6. *token_to_type*, *stype_to_string* e *gtype_to_string*: Utilizado para conversões entre tokens e tipos da gramática além de auxiliar as rotinas de impressão no console.

4.3 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. 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*

4.4 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 uma cadeia de caracteres que irá representar o escopo do símbolo (será preenchido posteriormente na análise semântica), o tipo do símbolo informando se é uma função ou variável, um identificador com o nome da variável ou função e para facilitar verificações qual o tipo do símbolo na gramática (*void*,

int, *graph*, *array* e etc) 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.

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. Chamada indevida caso os tipos instanciados para a chamada de função não possuam uma equivalência de tipos.
5. Incompatibilidade do tipo em *return* e tipo da função.
6. Incompatibilidade de tipos em operações de atribuição.

6 Arquivos de teste

6.1 Arquivos de teste léxico

Os arquivos *lexico/correto1.txt*, *lexico/correto2.txt*, *lexico/errado1.txt* e *lexico/errado2.txt* possuem exemplos de programas sem e com erros léxicos. Em *errado1.txt* são apresentados caracteres não reconhecidos pela linguagem, espera-se que sejam informados um a um os caracteres incorretos, e em *errado2.txt* 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.txt*, *sintatico/correto2.txt*, *sintatico/errado1.txt* e *sintatico/errado2.txt* possuem exemplos de programas sem e com erros sintáticos. Em *errado1.txt* 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.txt* 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.

6.3 Arquivos de teste semânticos

Os arquivos *semantico/correto1.txt*, *semantico/correto2.txt*, *semantico/errado1.txt* e *semantico/errado2.txt* 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.

```
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

```
void main() { }

void main () {      // Função redeclarada
    int a = 2;
    int a;          // Variável redeclarada
    {
        int a;
        outra();    // Função inexistente/fora de escopo
        b = 2;      // Variável inexistente/fora de escopo
        graph c = a;// Atribuição entre tipos distintos
    }
}
```

```
graph outra(int a) {
    return a;           // Retorno incompativel
}

void main() {
    graph g;
    int a = 1 + g;      // Operação aritmética com tipos incompativeis
    outra(g);           // Argumentos incompativeis
}
```

Figura 4. Programas comentados com os respectivos erros sintáticos

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 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.

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.