Linguagem para processamento 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 que se destina a área de processamento de grafos. A mesma tem como objetivo fornecer funcionalidades básicas de uma linguagem declarativa 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 a topologia de um grafo especificado.

2 Visão geral do tradutor

O tradutor implementado consiste em uma separação de processamento comumente utilizada no desenvolvimento de 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.

3 Gramática

3.1 Definição

$\langle program \rangle \quad ::= \langle program \rangle \ \langle function \rangle \mid$ ”

⟨*function*⟩ ::= ⟨*type*⟩ ⟨*id*⟩ '(' ⟨*opt-params*⟩ ')' '{' ⟨*statements*⟩ '}'

⟨*type*⟩ ::= 'boolean' | 'int' | 'float'
| 'void' | 'graph' '[' ⟨*number-int*⟩ ']'

⟨*opt-params*⟩ ::= '' | ⟨*params*⟩

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

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

⟨*statement*⟩ ::= ⟨*matched-statement*⟩ | ⟨*open-statement*⟩

⟨*matched-statement*⟩ ::= 'if' '(' ⟨*expr-decl*⟩ ')' ⟨*matched-statement*⟩ else ⟨*matched-statement*⟩
| ⟨*declaration-or-atrib*⟩ ';'
| ⟨*expr-decl*⟩ ';'
| ⟨*for*⟩
| ⟨*while*⟩
| 'return' ⟨*expr-decl*⟩ ';'
| ⟨*read*⟩
| ⟨*write*⟩
| ⟨*graph-add*⟩ ';'
| '{' ⟨*statements*⟩ '}'

⟨*open-statement*⟩ ::= 'if' '(' ⟨*expr-decl*⟩ ')' ⟨*statment*⟩
| 'if' '(' ⟨*expr-decl*⟩ ')' ⟨*matched-statement*⟩ else ⟨*open-statement*⟩

⟨*for*⟩ ::= 'for' '(' ⟨*declaration-or-atrib*⟩ ';' ⟨*expr-decl*⟩ ';' ⟨*expr-decl*⟩
')' ⟨*statement*⟩
| 'for' '(' ⟨*declaration*⟩ ':' ⟨*graph-op*⟩ ')' ⟨*statement*⟩

⟨*while*⟩ ::= 'while' '(' ⟨*expr-decl*⟩ ')' ⟨*statement*⟩

⟨*read*⟩ ::= '>>' ⟨*id-or-access*⟩ ';'

⟨*write*⟩ ::= '<<' ⟨*expr-decl*⟩ ';'

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

⟨*opt-params-call*⟩ ::= '' | ⟨*params-call*⟩

⟨*params-call*⟩ ::= ⟨*params-call*⟩ ',' ⟨*value*⟩ | ⟨*value*⟩

⟨*graph-add*⟩ ::= ⟨*id*⟩ '[' ⟨*expr-decl*⟩ 'to' ⟨*expr-decl*⟩ ']'

⟨*graph-op*⟩ ::= 'dfs' '(' ⟨*id*⟩ ',' ⟨*expr-decl*⟩ ',' ⟨*expr-decl*⟩ ')'
| 'bfs' '(' ⟨*id*⟩ ',' ⟨*expr-decl*⟩ ',' ⟨*expr-decl*⟩ ')'

⟨*declaration-or-atrib*⟩ ::= ⟨*declaration*⟩
| ⟨*declaration*⟩ '=' ⟨*expr-decl*⟩

⟨*declaration*⟩ ::= ⟨*type*⟩ ⟨*id*⟩

⟨*expr-decl*⟩ ::= ⟨*id-or-access*⟩ '=' ⟨*expr-relational*⟩ | ⟨*expr-relational*⟩

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

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

⟨*expr-or*⟩ ::= ⟨*expr-or*⟩ '||' ⟨*expr-add*⟩ | ⟨*expr-add*⟩

⟨*expr-add*⟩ ::= ⟨*expr-add*⟩ ⟨*add-op*⟩ ⟨*expr-mult*⟩ | ⟨*expr-mult*⟩

⟨*expr-mult*⟩ ::= ⟨*expr-mult*⟩ ⟨*mul-op*⟩ ⟨*expr-not*⟩ | ⟨*expr-not*⟩

⟨*expr-not*⟩ ::= ⟨*expr-unary*⟩ ⟨*factor*⟩ | ⟨*factor*⟩

⟨*expr-unary*⟩ ::= '!' | ⟨*add-op*⟩

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

⟨*value*⟩ ::= ⟨*id-or-access*⟩ | ⟨*number*⟩ | 'true' | 'false'

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

⟨*add-op*⟩ ::= '+' | '-'

⟨*mul-op*⟩ ::= '*' | '/'

⟨*number*⟩ ::= ⟨*number-int*⟩ | ⟨*add-op*⟩ ⟨*number-list*⟩ '.' ⟨*digit*⟩ ⟨*number-list*⟩

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

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

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

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

⟨*id*⟩ ::= ⟨*letter*⟩ ⟨*letter-list*⟩ ⟨*number-list*⟩

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

⟨*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. remoção de <add-op> da definição de number.
2. adição de regra <expr-unary> para números positivos e negativos.
3. adição de *true* e *false* em regra <value>

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 alguma função específica. 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.

1. Foi definido de melhor forma o funcionamento/uso da primitiva.
2. Foram feitas alterações gerais na escrita de sentenças longas por todo o documento, mas não foram acrescentadas nem removidas informações.

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

1. O ponto inicial de execução será sempre a função que possui o identificador de main e sua falta implica em não execução.
2. Parâmetros serão passados sempre por cópia assim como atribuições.
3. Cópia de variáveis do tipo *graph* apenas serão possíveis para aqueles que possuem um mesmo tamanho especificado entre parêntesis.
4. Cast implícito será possível entre *float* e *int*, e de *float* e *int* para *boolean*, caso valor seja diferente de zero será considerado *true*, e *false* caso valor seja zero.
5. Expressões de atribuição possuem um valor de retorno que será o do próprio identificador sendo atribuído.
6. O escopo de variáveis está será delimitado pelas chaves.
7. Grafo inicialmente não irá possuir arestas e possui valores de vértices default em zero.

1. Anteriormente não havia cast para boolean.

4 Descrição da analise léxica

Para a realização da análise léxica foi utilizado o gerador de analisador léxico *FLEX*. Foi realizada a definição regular para reconhecimento dos tokens para que em seguida fosse transportada para as definições em *FLEX*. Tokens foram definidos como uma estrutura em linguagem C para facilitar a utilização na fase de análise sintática. Em seguida foi definida uma política de tratamento de erros para esta fase da análise.

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.

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

4.2 Tokens

Cada símbolo terminal da gramática com exceção das regras derivadas de *id* e *number*, será atrelado a um identificador único como na figura 2 que irá determinar um token do analisador. Com relação aos números e identificadores os

mesmos devem além do identificador conter o valor do lexema que vai ser armazenado na estrutura que será informada pelo analisador. Também será necessário saber em que linha e coluna determinado token inicia para que sejam informados de forma correta erros léxicos e também posteriormente erros semânticos. Para que o token fosse extraído de acordo com a estrutura apresentada foi introduzida uma função no gerador para criação do token apresentado na figura 2.

```
enum {
    IF = 1, FOR, WHILE,
    BOOLEAN, INT, FLOAT, GRAPH, VOID,
    TO, TRUE, FALSE,
    ID, NUMBER,
    READ, WRITE, DFS, BFS,
    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
};

struct token {
    int id;
    int line, column;
    void* value;
};
```

Figura 2. Identificação de tokens e estrutura básica em C

4.3 Tabela de símbolos

A princípio não serão diretamente instanciadas informações da análise léxica na tabela de símbolos uma vez que o escopo da linguagem é definido por seções de código entre chaves, portanto a mesma será preenchida nos demais passos da tradução. A tabela será uma composição de tabelas onde cada função possui sua própria hierarquia de escopos. Será realizada desta forma para permitir a separação e correta designação de identificadores de acordo com a região de código que se encontram.

4.4 Identificação e tratamento de erros

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.

4.5 Arquivos de teste

Os arquivos *correto1.txt*, *correto2.txt*, *errado1.txt* e *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 3, 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 3. Programas comentados com os respectivos erros léxicos

5 Dificuldades da implementação

A implementação da nova funcionalidade talvez não seja tão complexa por parte do analisador sintático, porém adiciona complexidade na verificação semântica poís será necessário verificar atribuições entre grafos se atentando a definição para que seja efetuada uma operação válida. Outro problema pode se apresentar no momento de gerar um código intermediário uma vez que um grafo pode ser visto como uma matriz bidimensional que pode não ser diretamente ou facilmente implementável na linguagem de destino.

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.