Definição do tema: 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 e também 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 também poupa a necessidade do programador em manter e cuidar do uso de sua própria estrutura e ter que carrega-lá a todo momento que decidir iniciar um novo programa, permitindo que algum comportamento seja aplicado de forma prática a topologia de um grafo especificado.

2 Gramática

2.1 Definição

⟨*program*⟩	::=	⟨*program*⟩ ⟨*function*⟩ | ”
⟨*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*⟩ ::= '!' ⟨*factor*⟩ | ⟨*factor*⟩

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

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

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

⟨*add-op*⟩ ::= '+' | '-'

⟨*mul-op*⟩ ::= '*' | '/'

⟨*number*⟩ ::= ⟨*add-op*⟩ ⟨*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' | '_'

2.2 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.

Os comandos de *for*, *while* e *if* contemplam expressões que avaliaram 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]*. Existem duas operações associadas a um grafo, a operação de inserção de arestas e inserção de valores de armazenamento numérico com uso explicitado mais abaixo em código.

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 underline.

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 e procurando
        // arestas 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 apenas será possível entre *float* e *int*, portanto expressões lógicas não se misturam com aritméticas.
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.

3.1 Dificuldades da implementação

A implementação da nova feature talvez não seja tão complexa por parte do analisador sintático, porém adiciona complexidade na verificação semântica pois 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 lista de listas ou uma matriz bidimensional que podem não ser diretamente ou facilmente implementável na linguagem de destino. [ALSU07]

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.