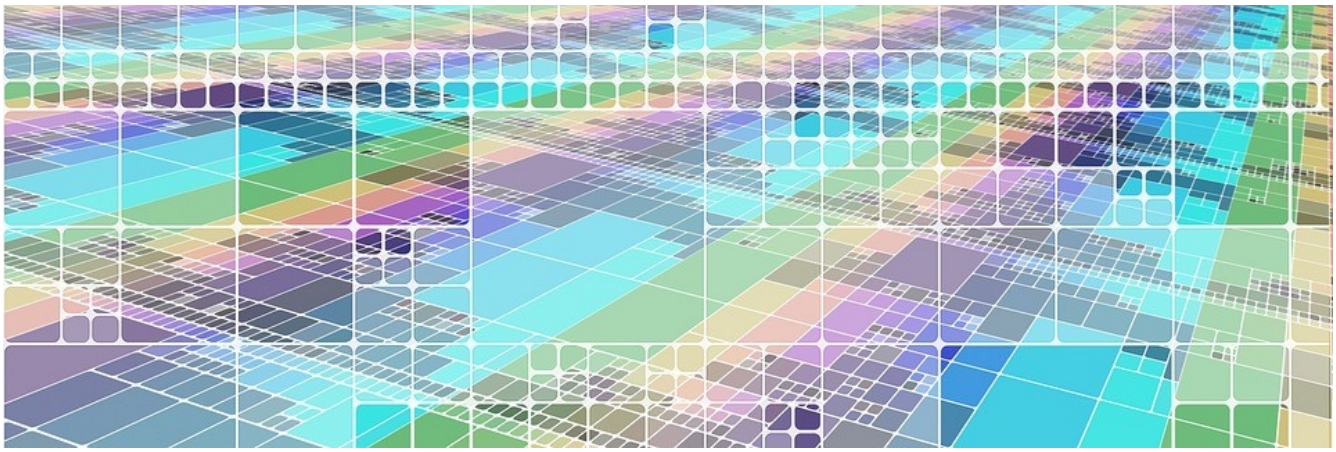
GOL: Graph Oriented Language.

4ª Edição



Michelangelo da Rocha Machado - 140156089
Universidade de Brasília, Departamento de Ciência da Computação.

SUMÁRIO

GRAMÁTICA

[topo]

A gramática da linguagem GOL foi construída com base na gramática ANSI C publicada por Jeff Lee [1]. Símbolos terminais estão entre *aspas duplas*. Notação utilizada: EBNF.

Regras léxicas

[topo]

```
letter      ::= [a-zA-Z]
digit       ::= [0-9]

Ident       ::= letter { letter | digit }
Integer     ::= digit+
String      ::= '"' str '"'
Double      ::= digit+ '.' digit+ ('e' '-'? digit+)?
str         ::= ? Qualquer ASCII imprimível que não seja 34 (") ou 10 (\n) ?
```

Regras Sintáticas

[topo]

```
TraUniExtVar.     Trans_Unit       ::= Ext_Var_Decl ;
TraUniList.       Trans_Unit       ::= Trans_Unit Ext_Var_Decl ;

AssOpEQ.          Assign_Operator  ::= "=" ;
AssOpINS.         Assign_Operator  ::= "<<" ;

ConstInt.         Constant         ::= Integer ;
ConstDouble.      Constant         ::= Double ;
ConstStr.         Constant         ::= String ;

UnaOpMinus.       Unary_Operator   ::= "-" ;
UnaOpNot.         Unary_Operator   ::= "!" ;

TypeVoid.         Type             ::= "void" ;
TypeInt.          Type             ::= "int" ;
TypeDouble.       Type             ::= "double" ;
TypeGraph.        Type             ::= "graph" ;

ArgExpListExp.    Arg_Exp_List     ::= Expression ;
ArgExpList.       Arg_Exp_List     ::= Arg_Exp_List "," Expression ;

PriExpId.         Primary_Exp      ::= Ident ;
PriExpConst.      Primary_Exp      ::= Constant ;
PriExpExp.        Primary_Exp      ::= "(" Expression ")" ;
```

```
PosExpPri.        Posfix_Exp        ::= Primary_Exp ;
PosExpSub.        Posfix_Exp        ::= Ident "[" Primary_Exp "]" ;
PosExpIn.         Posfix_Exp        ::= Ident "@" Primary_Exp "#" ;
PosExpOut.        Posfix_Exp        ::= Ident "#" Primary_Exp "@" ;
PosExpNeig.       Posfix_Exp        ::= Ident "&" Primary_Exp "&" ;
PosExpCal.        Posfix_Exp        ::= Ident "(" ")" ;
PosExpCalArg.     Posfix_Exp        ::= Ident "(" Arg_Exp_List ")" ;

UnaExpPos.        Unary_Exp         ::= Posfix_Exp ;
UnaExpOp.         Unary_Exp         ::= Unary_Operator Unary_Exp ;

MulExpUna.        Multi_Exp         ::= Unary_Exp ;
MulExpMul.        Multi_Exp         ::= Multi_Exp "*" Unary_Exp ;
MulExpDiv.        Multi_Exp         ::= Multi_Exp "/" Unary_Exp ;

AddExpMul.        Add_Exp           ::= Multi_Exp ;
AddExpAdd.        Add_Exp           ::= Add_Exp "+" Multi_Exp ;
AddExpSub.        Add_Exp           ::= Add_Exp "-" Multi_Exp ;

RelExpAdd.        Rel_Exp           ::= Add_Exp ;
RelExpLT.         Rel_Exp           ::= Rel_Exp "<" Add_Exp ;
RelExpGT.         Rel_Exp           ::= Rel_Exp ">" Add_Exp ;
RelExpLE.         Rel_Exp           ::= Rel_Exp "<=" Add_Exp ;
RelExpGE.         Rel_Exp           ::= Rel_Exp ">=" Add_Exp ;

EqExpRel.         Eq_Exp            ::= Rel_Exp ;
EqExpEQ.          Eq_Exp            ::= Eq_Exp "==" Rel_Exp ;
EqExpNE.          Eq_Exp            ::= Eq_Exp "!=" Rel_Exp ;

LogAndExpEq.      Log_And_Exp       ::= Eq_Exp ;
LogAndExpAnd.     Log_And_Exp       ::= Log_And_Exp "&&" Eq_Exp ;

LogOrExpLogAnd.   Log_Or_Exp        ::= Log_And_Exp ;
LogOrExpLogOr.    Log_Or_Exp        ::= Log_Or_Exp "||" Log_And_Exp ;

ExpLogOr.         Expression        ::= Log_Or_Exp ;
ExpAss.           Expression        ::= Ident Assign_Operator Expression ;
ExpAssGraph.      Expression        ::= Ident Assign_Operator "(" Expression ","
Expression ")" ;

IniDecListIni.    Init_Decl_List    ::= Init_Declarator ;
IniDecList.       Init_Decl_List    ::= Init_Decl_List "," Init_Declarator ;

IniDecId.         Init_Declarator   ::= Ident ;
IniDecIdE.        Init_Declarator   ::= Ident Assign_Operator Log_Or_Exp ;

VarDec.           Var_Declaration   ::= Type Init_Decl_List ";" ;

VarDecListVar.    Var_Decl_List     ::= Var_Declaration ;
VarDecList.       Var_Decl_List     ::= Var_Decl_List Var_Declaration ;
```

```
ParamListId.     Parameter_List   ::= Type Ident ;
ParamList.       Parameter_List   ::= Parameter_List "," Type Ident ;

StmListStm.      Statement_List   ::= Statement ;
StmList.         Statement_List   ::= Statement_List Statement ;

StmOpen.         Statement        ::= Open_Stm ;
StmClosed.       Statement        ::=  Closed_Stm ;

OpnStmIfSmp.     Open_Stm         ::= "if" "(" Expression ")" Simple_Stm ;
OpnStmIfOpn.     Open_Stm         ::= "if" "(" Expression ")" Open_Stm ;
OpnStmIfCls.     Open_Stm         ::= "if" "(" Expression ")" Closed_Stm "else"
Open_Stm ;
OpnStmWhile.     Open_Stm         ::= "while" "(" Expression ")" Open_Stm ;

BlkStm.          Block_Stm        ::= "{" "}" ;
BlkStmList.      Block_Stm        ::= "{" Statement_List "}" ;
BlkStmVar.       Block_Stm        ::= "{" Var_Decl_List "}" ;
BlkStmVarStm.    Block_Stm        ::= "{" Var_Decl_List Statement_List "}" ;

RetStmRet.       Return_Stm       ::= "return" ";" ;
RetStmExp.       Return_Stm       ::= "return" Expression ";" ;

ExpStmNul.       Exp_Stm          ::= ";" ;
ExpStmExp.       Exp_Stm          ::= Expression ";" ;

ClosedStmSmp.    Closed_Stm       ::= Simple_Stm ;
ClosedStmIf.     Closed_Stm       ::= "if" "(" Expression ")" Closed_Stm "else"
Closed_Stm ;
ClosedStmWhile.  Closed_Stm       ::= "while" "(" Expression ")" Closed_Stm ;

SmpStmBlock.     Simple_Stm       ::= Block_Stm ;
SmpStmExp.       Simple_Stm       ::= Exp_Stm ;
SmpStmRet.       Simple_Stm       ::= Return_Stm ;

DecIdParam.      Declarator       ::= Ident "(" Parameter_List ")" ;
DecId.           Declarator       ::= Ident "(" ")" ;

FunDef.          Function_Def     ::= Type Declarator Block_Stm ;

ExtVarDecFun.    Ext_Var_Decl     ::= Function_Def ;
ExtVarDecVar.    Ext_Var_Decl     ::= Var_Declaration ;
```

Associatividade e precedência

Operador	Associatividade
!, – (unário)	direita para esquerda
*, /	esquerda para direita
+, – (binário)	esquerda para direita
<, <=, >, >=	esquerda para direita
==, !=	esquerda para direita
&&	esquerda para direita
||	esquerda para direita

Efeito de algumas operações

As operações a seguir foram derivadas de algumas das operações básicas encontradas na biblioteca Ngraph [2] para lidar com grafos em *C++*.

```
graph G1;          //Declaração de variável do tipo graph
graph G2;
graph G3;
int a = 13;

G1 << a;           //Inserção de vértice 13 no grafo G1
G1 << (a,14);      //Inserção de aresta (13,14) no grafo G1
G2 << 15;          //Inserção de vértice 15 no grafo G2

G3 = G1 + G2;      //G3 recebe união dos grafos G1 e G2

G2 = G3[G1];       //G2 recebe subgrafo de G3 composto por vértices de G1
a  = G3@13#;       //a recebe o grau de entrada do nó 13
a  = G3#13@;       //a recebe o grau de saída do nó 13
G3 = G1&13&;       //G3 recebe grafo composto pelos vizinhos do nó 13 no grafo G1
```

SEMÂNTICA

[topo]

A seguir são descritos alguns pontos relacionados à semântica da linguagem GOL. A linguagem possui escopo estático e também é estáticamente tipada.

1 Regras de escopo

Em GOL, uma variável deve ser declarada antes de poder ser utilizada. Mais do que isso, todas as declarações devem ser realizadas na parte inícial de um bloco de declarações. Também se faz necessário que o código que faz referência a um nome ocorra após a declaração de tal nome. Por fim, não são permitidas redefinições de um mesmo nome em um mesmo escopo.

Além do escopo global, toda estrutura de bloco resulta no aninhamento de um novo escopo, o que resulta na possibilidade de uma quantidade indefinida de escopos, de forma que a visibilidade de uma variável se restringe apenas a referências feitas nos escopos mais internos àquele no qual a variável foi declarada. Destaca-se também que funções só podem ser declaradas no escopo global.

2 Funções

Os argumentos em uma chamada de função devem coincidir em número e tipo com relação aos parâmetros da função chamada.

3 Conversões de tipos

A conversão de tipos só é feita entre “int” e “double”. Tal conversão é feita de maneira implícita. Inteiros não podem ser utilizados em locais que esperam tipos booleanos, como “if” ou “while”.

4 Operadores e tipos

Aqui é feita uma breve descrição da compatibilidade entre os operadores e os tipos de dados para cada operação. Para as operações aplicadas aos tipos numéricos e booleanos, assume-se a semância tradicional das operações, para o caso das operações sobre grafos, é feita uma descrição a respeito da semântica, bem como dos tipos utilizados em cada operação, destacados em amarelo.

Numéricos

Para os tipos númericos (“int” e “double”), as operações permitidas são:

- (unário e binário), *, /, +, -, <, <=, >, >=, ==, !=

Booleanos

Para o tipo “bool”, as operações permitidas são:

!, ==, !=, && e ||

Grafos

Para o tipo “graph”, as operações permitidas são:

[graph] << [int] Inserção de vértice, informado pela expressão de tipo “int”, em variável do tipo “graph”.

[graph] << ($[int_1]$, $[int_2]$) Inserção de aresta, informada pelas expressões de tipo “int”, em variável do tipo “graph”.

$[graph_1]$ [($[graph_2]$)] Retorno de um valor do tipo “graph” composto pelos vértices em comum das duas variáveis referenciadas.

[graph] @([int])# Retorno de um valor do tipo “int”, contendo o grau de entrada do nó cujo valor está contido na expressão de tipo “int” passada.

[graph] #([int])@ Retorno de um valor do tipo “int”, contendo o grau de saída do nó cujo valor está contido na expressão de tipo “int” passada

[graph] &([int])& Retorno de um valor do tipo “graph” contendo somente os nós vizinhos do nó, cujo valor está contido na expressão de tipo “int” passada, dentro do grafo referenciado.

5 Declarações de retorno

Todas as funções devem possuir uma declaração de retorno cujo tipo da expressão deve ser compatível com o tipo existente na declaração da função, com exceção de funções declaradas com tipo “void”, nas quais é dispensado o uso de uma declaração de retorno, mas no caso de haver tal declaração, essa não pode conter uma expressão cujo tipo seja diferente de “void”.

6 Programa mínimo

Um programa na linguagem GOL deve possuir pelo menos uma declaração, seja de função, seja de variável, para ser considerado válido.

Notas

Os únicos tipos numéricos são: ‘int’ e ‘float’; Não existe sobrecarga de nomes de funções; Em um ‘if’, o ‘else' estará associado com o ‘if’ mais recente; Expressões matemáticas são avaliadas em ordem consistente com aquelas da matemática; O tipo grafo armazena grafos direcionados sem repetição de arestas; O tamanho máximo de um identificador é 31 caracteres; As declarações de variáveis devem constar antes dos statements;

POLÍTICA DE TRATAMENTO DE ERROS

[topo]

Erros tratados:

1 Léxico

String literal não fechada.

Descrição:

Uma string literal foi aberta e o analisador léxico encontrou um ‘EOL’ ou ‘EOF’ antes que a string fosse fechada.

Tratamento:

Um caractere para fechar a string é inserido antes do fim de linha e a análise léxica continua como se a string compreendesse toda a linha, possibilitando a identificação de mais erros. Como o analisador empilha a posição das strings abertas e desempilha somente quando encontra um caractere de fim de string literal.

Comentário não fechado.

Descrição:

Um comentário de múltiplas linhas foi aberto e o analisador léxico encontrou um ‘EOF’ antes que o comentário fosse fechado.

Tratamento:

O analisador sempre guarda a posição do último comentário de múltiplas linhas aberto e libera a memória quando este é fechado. Portanto, no caso de encontrar o ‘EOF’, é relatada a posição do último comentário que foi aberto e que nunca foi fechado.

Tamanho de identificador excedeu o limite.

Descrição:

Um identificador foi declarado com tamanho superior à 31 caracteres, excedendo o tamanho máximo permitido.

Tratamento:

Nesse caso é emitido um Warning de que o identificador foi truncado e a análise continua como se o identificador tivesse sido declarado com o tamanho truncado de 31 caracteres.

Token não definido.

Descrição:

Um token não definido foi identificado pelo analisador.

Tratamento:

A posição, bem como o caractere em si são relatados pelo analisador no detalhamento do erro.

2 Sintático

[topo]

A política para os erros a seguir seguiu a descrição do procedimento para recuperação de erros do manual do *Bison* [3], aonde algumas regras específicas foram tratadas conforme o token ausente.

Declarator

Descrição:

- Ausência de identificador em uma chamada à função.

Closed_Stm

Desrição:

- Ausência de expressão dentro de *if.*
- Ausência de expressão dentro de *while.*

Open_Stm

Descrição:

- Ausência de expressão dentro de *if.*
- Ausência de expressão dentro de *while.*

Var_Declaration

Descrição:

- Ausência de identificador na declaração de uma variável com inicialização.

Expression

Descrição:

- Ausência de operador de atribuição em uma expressão de atribuição.

Posfix_Exp

Descrição:

- Ausência de uma expressão dentro do operador de subgrafo para um grafo.
- Ausência de uma expressão dentro do operador de grau de entrada para um nó.
- Ausência de uma expressão dentro do operador de grau de saída para um nó.
- Ausência de uma expressão dentro do operador de vizinhança para um nó.

3 Semântico

[topo]

O tratamento de erros semânticos considera os seguintes problemas. Em todos os casos, os cenários descritos geram a reportação de um erro de compilação.

Referência sem declaração

Ao se detectar a referência ou definição de um nome qua não foi préviamente declarado.

Redeclaração

Ao se declarar mais de uma vez um nome em um mesmo escopo.

Conversão entre tipos inválidos

Qualquer tentativa de converter tipos que não seja entre “int” e “double”

Chamada de função com argumentos inválidos

Qualquer chamada de função que não corresponda com número ou tipo dos parâmetros definidos na declaração da função.

Retorno de tipo incorreto

Qualquer declaração de retorno cujo tipo da expressão não corresponda ao tipo definido na declaração da função.

Atribuiçao inválida

Qualquer atribuição cujos tipos sejam diferentes (com exceção dos casos em que ocorre conversão entre os tipos “int” e “double”).

Operação com tipo inválido

Qualquer operação que não respeite o valor semântico descrito.

Declaração feita no corpo de um bloco

Qualquer declaração de variável feita após a ocorrência de uma outra declaração qualquer que não seja declaração de variáveis (e.g. atribuição).

FUNÇÕES E ESTRUTURAS

[topo]

```
void count();
```

Descrição:
Conta o número de linhas e colunas consumidas da entrada. Tab está definido como TAB_LENGTH espaços.

```
void yyerror(const char *str);
```

Descrição:
Foi definida para fornecer informações adicionais sobre a coluna atual e o token do local aonde foi detectado o erro.

Demais funções:

- Foram adicionadas funções para realizar a impressão da árvore sintática, aonde, para cada regra foi adicionada uma função que imprime a árvore de acordo com a estrutura da regra. Após a edição 4 desse relatório, a função oferece a possibilidade de gerar um arquivo de extensão “dot” e correspondente “png” com uma representação gráfica da árvore abstrata.
- Foram adicionadas funções para a construção da árvore sintática. Após a refatoração dos nós da árvore, a função de construção de nós é genérica.
- Foram adicionadas funções para inserção de elementos na tabela de símbolos bem como para a criação desta. Também foram adicionadas funções para realizar a busca de um símbolo na tabela, bem como funções para sua impressão no terminal.

Tabela de símbolos

A tabela de símbolos foi implementada como uma lista encadeada, devido à simplicidade do design e a sua aceitação como estrutura válida para tal propósito [4]. A tabela possuí as seguintes informações: Valores constantes, identificadores, lista de parâmetros de funções, tags de identificação e linha e coluna do código fonte.

Árvore abstrata

Adaptando a estrutura proposta nas notas de aula do curso de compiladores da universidade do Tennessee [5], foram adicionadas estruturas do tipo struct para os representar os nós da árvore abstrata, aonde uma tag foi utilizada para diferenciar cada possível nome de regra. O único campo relativo à árvore abstrata anotada foi o tipo do nó.

Demais estruturas:

- Foi criada uma estrutura para representar os filhos de um nó, o qual é uma lista encadeada simples.
- Foi criada uma estrutura para reunir os parâmetros de uma função dentro da tabela de símbolos.
- Foi criada uma estrutura para gerenciar todas as referências para as tabelas de símbolos criadas durante a compilação.

ARQUIVOS DE TESTE

[topo]

Somente léxico

Esses arquivos estão localizados na pasta “Arquivos de teste” → “Lexico”

1 Arquivos com código correto

`in_ok_1.txt`
`in_ok_2.txt`

2 Arquivos com código incorreto

`AllTogether.txt`
Erros:
Comentário não fechado. Linha: 8, Coluna: 8.
Identificador com tamanho máximo excedido. Linha: 5, Coluna: 9.
String Literal não fechada. Linha: 3, Coluna: 18.
Token não identificado. Linha: 3, Coluna: 17.

`in_1.txt`
Erros:
If sem expressão. Linha: 3, Coluna: 8.
If sem expressão. Linha: 15, Coluna: 8.

`in_2.txt`
Erros:
If sem expressão. Linha: 3, Coluna: 8.
While sem expressão. Linha 15; Coluna 11.

`in_3.txt`
Erros:
Sem tipo na decla
String Literal não fechada. Linha: 3, Coluna: 22.

`in_4.txt`
Erros:
Token não identificado. Token: $, Linha: 5, Coluna: 5.
Token não identificado. Token: _, Linha: 4, Coluna: 17.

Léxico e sintático

Esses arquivos estão localizados na pasta “Arquivos de teste” → “Sintatico”

1 Arquivos com código correto

`in_ok_1.txt`
`in_ok_2.txt`

2 Arquivos com código incorreto

`AllTogether.txt`
Erros:
Declaração de variável sem identificador. Linha 3; Coluna 9.
Declaração de variável sem identificador. Linha 12; Coluna 8.
While sem expressão. Linha 17; Coluna 11.
If sem expresão. Linha 4; Coluna 8.

`in_1.txt`
Erros:
Comentário não fechado. Linha: 5, Coluna: 5.

`in_2.txt`
Erros:
Identificador com tamanho máximo excedido. Linha: 4, Coluna: 11.
Identificador com tamanho máximo excedido. Linha: 3, Coluna: 9.

`in_3.txt`
Erros:
Declaração de variável sem identificador. Linha: 7, Coluna: 8.

PROBLEMAS

[topo]

1 Memory leak

Infelizmente não houve tempo para a construção das estruturas responsáveis por liberar a memória alocada na construção da árvore e da tabela de símbolos. Portanto, como era previsto, o uso da ferramenta valgrind reportou a existência de áreas de memória não desalocadas, porém, nenhuma das referências é perdida, indicando que o problema de fato parece se restringir à liberação da árvore e tabela de símbolos.

2 Exibição da árvore

A notação utilizada pode se tornar confusa para grandes entradas e, consequentemente, pouco útil na depuaração. Por essa razão, foi incorporado um mecanismo nas funções relacionadas à impressão para a geração de um arquivo “dot” com o respectivo “png”, o qual pode ser gerado seguindo-se as instruções fornecidas durante a execução.

3 Análise semântica

Infelizmente, devido a um bug identificado algumas poucas horas antes do prazo de entrega, a submissão do código relativo ao semântico não considerou a marioria dos cenários de análise descritos aqui. Entretanto, todas os mecanismos citados foram, sem exceção, concluídos posteriormente (cerca de 6 hroas após o prazo de entrega). Portanto na 5ª edição desse documento todos os mecanismos citados nesse relatório estarão presentes.

REFERÊNCIAS

[topo]

Bibliografia

1: Jeff Lee, ANSI C grammar, 1985, https://www.lysator.liu.se/c/ANSI-C-grammar-l.html
2: Roldan Pozo, NGrapha simple (Network) Graph library in C++, ,
https://math.nist.gov/~RPozo/ngraph/
3: , Bison 3.4 - Simple Error Recovery, 2019,
https://www.gnu.org/software/bison/manual/html_node/Simple-Error-Recovery.html
4: Bradley T. Vander Zanden, , 2011,
http://web.eecs.utk.edu/~bvanderz/teaching/cs461Sp11/notes/parse_tree/
5: Robert Sedgewick, Algorithms (4th Edition), 2009