**UFSJ –** São João Del Rei

Lista 1

Compiladores

Marcelo Ari de Oliveira

Gabriel Rozante Oliveira

1. O código não apresenta nenhum erro que o analisador léxico possa reconhecer. Os erros presentes são erros sintáticos.
2. Há apenas um erro léxico presente, que é o $, pois não pertence a linguagem. Além desse, há mais alguns erros, porém são sintáticos. Esse erro está presente na linha “float 2a;” onde a linguagem não aceita uma variável começando com um número.
3. A) Sim , Existem conflitos, pois existe mais de uma possibilidade para a árvore de derivação para uma determinada entrada.

B) Isso se da devido ao fato que todos os operadores são gerados a partir de uma mesma produção, com isso a ordem de precedência vai depender apenas da ordem em que aparecer

C) E -> TE’

E’ -> + TE’ | - TE’ | T | ε

F-> \*TE’ | F’

F’-> ^TE’

T -> id | num

Assim ela irá considerar a precedência de maneira correta e sem conflitos.

1. Pilha | restante da entrada | Ação

E | id - float\_const / integer\_const $ | Desempilha e empilha E’ T

E’ T | id - float\_const / integer\_const $ | Desempilha e empilha T’ F

E’ T’ F | id - float\_const / integer\_const $ | Desempilha e empilha id

E’ T’ id | id - float\_const / integer\_const $ | Desempilha e avança na entrada E’ T’ | - float\_const / integer\_const $ | Desempilha (ε)

E’ | - float\_const / integer\_const $ | Desempilha e empilha E’ T –

E’ T- | - float\_const / integer\_const $ | Desempilha e avança na entrada E’ T | float\_const / integer\_const $ | Desempilha e empilha T’ F

E’ T’ F | float\_const / integer\_const $ | Desempilha e empilha float\_const

E’ T’ float\_const | float\_const / integer\_const $ | Desempilha e avança na entrada

E’ T’ | / integer\_const $ | Desempilha e empilha T’ F /

E’ T’ F / | / integer\_const $ | Desempilha e avança na entrada

E’ T’ F | integer\_const $ | Desempilha e empilha integer\_const

E’ T’ integer\_const | integer\_const $ | Desempilha e avança na entrada

E’ T’ | $ | Desempilha (ε)

E’ | $ | Desempilha (ε)

ε | $ | Aceita a entrada

Void funçao(){

If(tokken == function){

Math(id);

Math(function);

Math(()); ComandSeq();

Math (endfunction);

}

else{

Erro sintatico;

}

}

Void ComandoSeq(){

Comando();

Math (;);

ComandoSeq();

}

Void Comando(){

If(tokken == id){

Math (id);

Comando2();

}

else {

Erro Sintatico;

}

}

Void Comando2(){

If(tokken = ()){

Math (());

}

else if(tokken == (=)){

Math(=);

Expr();

}

else{

Erro Sintatico;

}

}

Void Expr (){

If(tokken == []){

Math([]);

Para\_Expr();

}

else {

Erro Sintatico;

}

}

Void Para\_Expr(){

If(tokken == integer\_const){

Math(integer\_const);

Param\_Expr2();

}

else {

Erro Sintatico;

}

}

Void Param\_Expr2(){

Math(:);

If(tokken == integer\_const){

Param\_Expr2();

}

}

6 –

Funções necessárias:

Preciso de 2 funçoes uma para passar 2 parametros para verificar a compatibilidade entre eles e outra parar passar o id\_lexema e verificar seu tipo logo temos:

verificacompatibilidade(tipo1,tipo2){

If (tipo1 == invalido) return invalido;

elif (tipo2 == invalido) return invalido;

elif(tipo1 == int && (tipo2 == char || tipo2 == short))

Erro Semantico;

return Invalido;

}elif(tipo2 == int && (tipo1 == char || tipo1 == short))

Erro Semantico;

return Invalido;

elif(tipo1 == char &&(tipo2 == int || tipo2 ==float)

Erro Semantico;

return Invalido;

elif(tipo2 == char &&(tipo1 == int || tipo1 == float)

Erro Semantico;

return Invalido;

elif(tipo1 == float && (tipo2 == char || tipo2 == short))

Erro Semantico;

return Invalido;

elif(tipo2 == float && (tipo1 == caractere || tipo1 == short))

Erro Semantico;

return Invalido;

}

Tabsimbolos.getTipo(id){

If id in tabsimbolos

return id.tipo;

}

Iremos definir as regras necessárias:

R1-> F.tipo = int

R2-> F.tipo = float

R3->F.tipo = tabsimbolos.getTipo(id.lexema)

R4-> F.tipo = F.tipo

R5-> T’.tipo = F.tipo

R6->T’.tipo = verificacompatibilidade(T’.tipo,F.tipo)

R7->E’.tipo = T.tipo

R8-> E’.tipo = verificacompatibilidade(E’Tipo,T.tipo)

R9-> t=tabsimbolos.getTipo(id.lexema)

R10-> r= verificacompatibilidade(t,E.tipo)

R11-> T.tipo = T’.Tipo

R12-> E.tipo = E’.tipo

Adicionando as Regras a nossa produção:

Assign -> [ R9 ] id = E [ R10 ];

AssignList AssignList ->[ R9 ] id = E [R10] ; AssignList | ε

E -> T [ R7 ] E' [R12]

E'-> + T [ R8 ] E' | - T [ R8 ] E' | ε

T -> F [R5] T' [ R11 ]

T' -> \* F [ R6 ] T' | / F [ R6 ] T' | ε

F -> ( E [ R6 ])

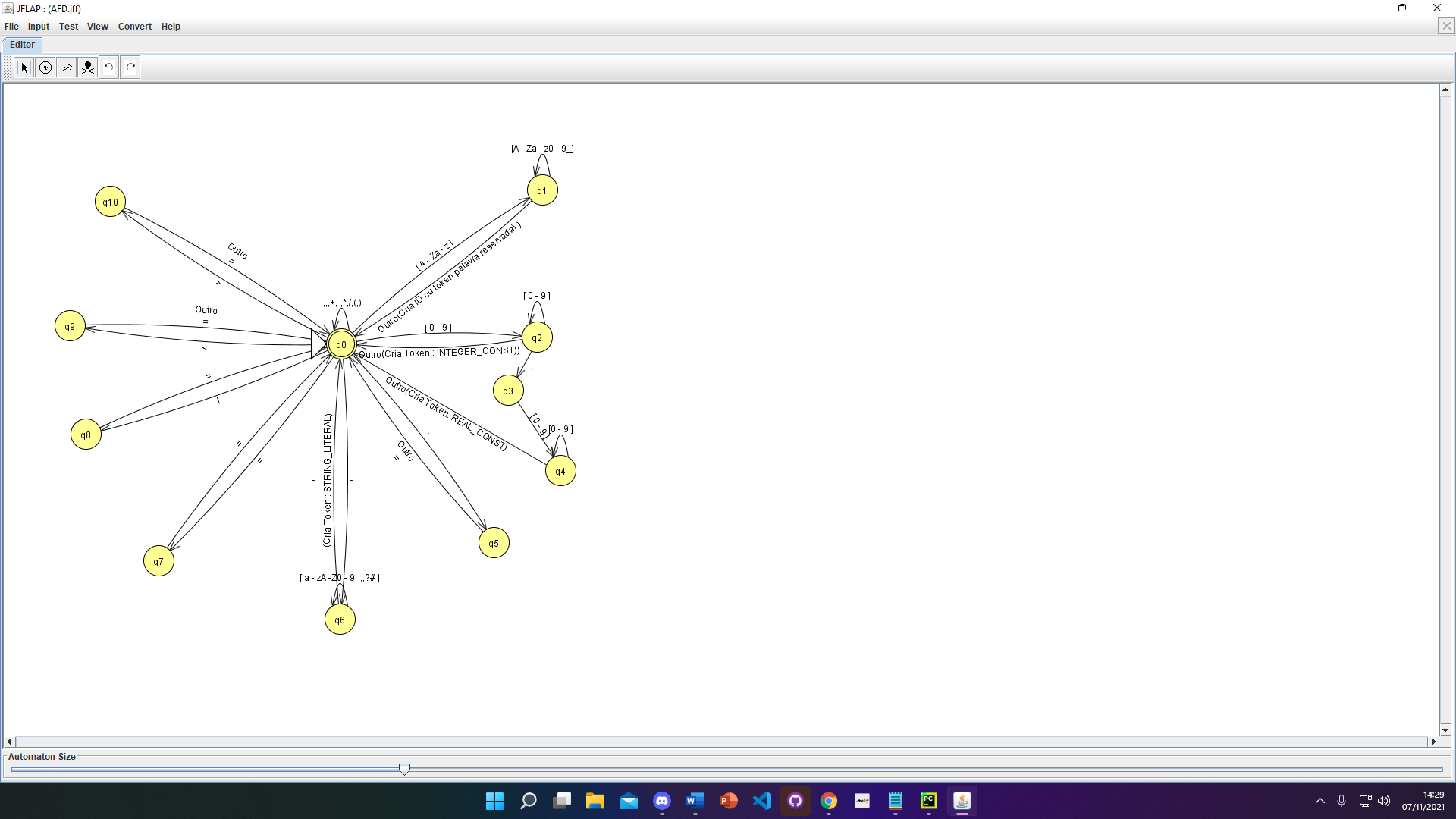
F -> [ R1 ] integer\_const

F -> [ R2 ] float\_const

F -> [ R3 ] id

7 –

a) Autômato (AFD):



b) O analisador léxico foi implementado da seguinte maneira, primeiramente recebemos o nome do arquivo que será aberto para leitura dos dados, logo após com base no autômato definimos os 10 estados (que foi a quantidade de estados que encontramos fazendo o autômato) no código e colocamos estes estados dentro de IF’s e elses, além disso criamos alguns vetores , um para a lista de tokens identificados, um pro buffer que é utilizado quando estamos formando o lexema e um terceiro que irá conter o token o lexema e a linha.

Implementamos o analisador léxico com a ideia de que caso um dado de entrada não esteja no nosso dicionário, acontecerá um erro léxico.

A princípio essa parte do código não foi tão difícil de implementar, encontramos alguns problemas somente na parte da string\_literal, porque ao ler um caractere que não estava no dicionário aceitava como string literal, agora primeiramente ele dá o erro e em seguida mostra como ficaria a string literal.

Após corrigir o problema o analisador léxico esta funcionando perfeitamente.

c)

|  |  |  |
| --- | --- | --- |
|  | FIRST | FOLLOW |
| Programa | {program} | {$} |
| Bloco | {ε,var,begin} | {$} |
| DeclaracaoSeq | {var, ε } | {begin} |
| Declaracao | {var} | {begin,var} |
| VarList | {id} | {:} |
| VarList2 | {, , ε } | {:} |
| Type | {boolean, integer,real,string} | {;} |
| ComdandoSeq | {id,if,while,print,read, ε } | {end} |
| Comando | { id,if,while,print,read } | {id,end,if,while,print,read} |
| Expr | {id,INTEGER\_CONST,REAL\_CONST,  TRUE,FALSE,STRING\_LITERAL,(} | {;,then,do,)} |
| ExprOpc | {==, !=, id, ε} | {;,then,do,)}} |
| OpIgual | {==,!=} | {id,INTEGER\_CONST,REAL\_CONST,  TRUE,FALSE,STRING\_LITERAL,(} |
| Rel | {id,INTEGER\_CONST,REAL\_CONST,  TRUE,FALSE,STRING\_LITERAL,(} | {;, then , do,==,!=,)} |
| RelOpc | { ε , < , <= , > ,>=} | {;, then , do,==,!=,)}} |
| OpRel | {<, <= , > , >=} | {id,INTEGER\_CONST,REAL\_CONST,  TRUE,FALSE,STRING\_LITERAL,( } |
| Adicao | {id,INTEGER\_CONST,REAL\_CONST,  TRUE,FALSE,STRING\_LITERAL,(} | {;,then,do,==,!=,<,<=,>,>=,)} |
| AdicaoOpc | {+ , - , ε } | {;,then,do,==,!=,<,<=,>,>=,)} |
| OpAdicao | {+, - } | {id,INTEGER\_CONST,REAL\_CONST,  TRUE,FALSE,STRING\_LITERAL,(} |
| Termo | {id,INTEGER\_CONST,REAL\_CONST,  TRUE,FALSE,STRING\_LITERAL,(} | {;,then,do,==,!=,<,<=,>,>=,+,-,)} |
| TermoOpc | {\*, /, ε } | {;,then,do,==,!=,<,<=,>,>=,+,-,)} |
| OpMult | {\*, /} | {id,INTEGER\_CONST,REAL\_CONST,  TRUE,FALSE,STRING\_LITERAL,( } |
| Fator | {id,INTEGER\_CONST,REAL\_CONST,  TRUE,FALSE,STRING\_LITERAL,( } | {; , then, do,==, !=, <, <= ,> ,>= ,+ ,-,\*,/,)} |

d)

O analisador sintático foi implementado por meio de descida recursiva, a princípio é passado pro analisador sintático um vetor que teve sua origem no analisador léxico, a partir daí começa a verificação, levamos em conta para fazer o sintático, o FIRST e o FOLLOW, e a produção.

Para cada produção não terminal foi implementada uma função e pra cada terminal, há um match que verifica se o token que eu estou verificando é o token esperado de acordo com a produção.

Alguns problemas na implementação foram encontrados , primeiramente o analisador não conseguia voltar a uma função, por exemplo, ele passava por uma declaração de variável e supondo que encontrasse outra variável na frente, ele não conseguia retornar para analisar essa outra variável, isso aconteceu devido a um problema na função do varlist2 que foi solucionado, em certo momento o analisador também não conseguia continuar a analise depois do begin entretanto isso também foi resolvido e o código está funcionando.

e) Quanto a analise semântica e tabela de símbolos, primeiramente listo todas as variáveis na tabela de símbolos juntamente com seu tipo e a linha que se encontra, a partir daí, para identificar uma incompatibilidade na expressão ou uma variável que está sendo redeclarada ou que não foi declarada, consulto minha tabela de símbolos de forma que , na expressão tenho uma variável “compara”, que armazena o tipo da variável que antecede o sinal de ATTR ,e depois comparo esse tipo com os tipos que vem em seguida, se por exemplo o tipo for boolean e a variável que segue for string dá erro de compatibilidade , se apareceu um variável no programa e ela não esta na tabela de símbolos a variável não foi declarada.

A maior dificuldade foi encontrar o local onde tínhamos que criar a tabela de símbolos além disso, onde fazer a comparação das expressões, também foi bem difícil de definir, mas agora o código está funcionando.