

Compiladores  
IME-UERJ  
Bacharelado em Ciência da Computação

## **ANÁLISE SINTÁTICA DO PROJETO**

**Aluno: Fernando da Silva Estácio**  
**Professora: Lis Ingrid Roque**

**Data: 03/09/2022**

## 1. BASE TEÓRICA PARA IMPLEMENTAÇÃO DO COMPILADOR

Para fazer o compilador ser capaz de fazer a análise sintática de um código, dada a linguagem entregue para o projeto, foi necessário analisar essa linguagem para encontrar possíveis obstáculos para uma análise derivada mais a esquerda. Nessa linguagem, foram encontrados 3 problemas, uma recursão à esquerda na variável não-terminal *Exp*, como mostra a imagem abaixo, *Stmt* não está completamente fatorada à esquerda e uma recursão indireta pelas variáveis *PrefixExp* e *Var*.

```
Exp ::= Exp BinOp Exp
      | not Exp
      | - Exp
      | PrefixExp
      | Function
      | { (Field (, Field)* )opt }
      | nil
      | true
      | false
      | Number
      | String
```

Corrigindo essa recursão, foi criada um *Exp* auxiliar, chamada de *E*, e então a linguagem ficou da seguinte forma:

*Exp* -> **not** *Exp* *E* | - *Exp* *E* | *PrefixExp* *E* | *Function* *E* | { (*Field* (, *Field*)\* )<sup>opt</sup> } *E* | **nil** *E* | **true** *E* | **false** *E* | *Number* *E* | *String* *E*

*E* -> *BinOp* *Exp* | Vazio

```

Stmt
 ::=  Vars = Exps
      |  Function ...
      |  do Block end
      |  while Exp do Block end
      |  if Exp then Block (elseif Exp then Block)* (else Block)opt end
      |  return Expsopt
      |  break
      |  for Name = Exp , Exp ( , Exp)opt do Block end
      |  for Names in Exps do Block end
      |  localopt function Name FunctionBody
      |  local Names = Exps

```

Já para fatorar *Stmt*, foram criadas variáveis auxiliares *S1*, *S2* e *S3*, e *Stmt* foi modificada de acordo, para essa mudança:

```

Stmt -> [tudo igual até o primeiro For...] | for S1 | function Name FunctionBody
| local S3

S1 -> Name S2

S2 -> = Exp , Exp ( , Exp)opt do Block end | , Names in Exps do Block end | in
Exps do Block end

S3 -> function Name FunctionBody | Names = Exps

```

Por fim, a recursão indireta em *PrefixExp* e *Var*:

```

PrefixExp
 ::=  Var
      |  ( Exp )

Var
 ::=  Name
      |  PrefixExp [ Exp ]

```

Ela pode ser solucionada, passando os parâmetros que podem ser alcançados pelo *Prefix* para ele mesmo, e resolvendo a recursão a esquerda que surgirá (*PrefixExp* -> *Name* | *PrefixExp* [*Exp*] | ( *Exp* ), logo tomando o seguinte formato:

PrefixExp -> Name P | ( Exp ) P

P -> [ Exp ] P | Vazio

### 1.1. CÁLCULO DOS CONJUNTOS FIRST E FOLLOW PARA CRIAÇÃO DA TABELA PREDITIVA

Agora, antes de começar a implementação da análise preditiva, é necessário calcular os conjuntos First e Follow de cada elemento da linguagem, para que se possa construir a tabela que define a tomada de decisões do analisador sintático. Os conjuntos calculados serão apresentados na tabela a seguir:

Variaveis	First	Follow
<b>Block</b>	Name, (, function, do, while, if, return, break, for, local,\$	Elseif, else, end
<b>Stmt</b>	Name, (, function, do, while, if, return, break, for, local	;
<b>S1</b>	Name	;
<b>S2</b>	=, virgula, in	;
<b>S3</b>	function, Name	;
<b>Exps</b>	not, -, Name,(, function, {, nil, true, false, Number, String	; , do
<b>Exp</b>	not, -, Name,(, function, {, nil, true, false, Number, String	; , ], do, then, virgula, ), }, First(BinOp)
<b>E</b>	or, and, <, >, <=, >=, ~=, ==, .. , +, -, *, /, ^, Vazio	; , ], do, then, virgula, ), }, First(Exp)
<b>PrefixExp</b>	Name, (	; , [, ], do, then, virgula, ), }, First(BinOp)
<b>P</b>	[, Vazio	; , [, ], do, then, virgula, ), }, First(BinOp)
<b>Field</b>	[, Name	virgula, }
<b>BinOp</b>	or, and, <, >, <=, >=, ~=, ==, .. , +, -, *, /, ^	not, Name, (, function, {, nil, true, false, Number, String
<b>Vars</b>	Name, (	=
<b>Var</b>	Name, (	First(BinOp), do, then, virgula, ), ], ;, }, =



[illegible][illegible]

	;	Name	Number	String	{	}	(	)	[	]	\$
Block ->		(Stmt)*					(Stmt)*				(Stmt)*
Stmt ->		Vars = Exps					Vars = Exps				
S1 ->		Name S2									
S2 ->											
S3 ->		Names = Exps									
Exps ->		Exp ( , Exp )*	Exp ( , Exp )*	Exp ( , Exp )*	Exp ( , Exp )*		Exp ( , Exp )*				
Exp ->		PrefixExp E	Number E	String E	{ (Field ( , Field) *)opt} E		PrefixExp E				
E ->		Vazio	Vazio	Vazio	Vazio	Vazio	Vazio	Vazio		Vazio	
PrefixExp ->		Name P					( Exp ) P				
P ->	Vazio					Vazio		Vazio	[ Exp ] P	Vazio	
Field ->		Name = Exp							[ Exp ] = Exp		
BinOp ->											
Vars ->		Var ( , Var )*					Var ( , Var )*				
Var ->		Name					PrefixExp [ Exp ]				
Function ->											
FunctionBody ->		name ( Params opt ) Block end									
Param ->		Names									
Names ->		Name ( , Name )*									

## 1.2. PROBLEMAS EXTRAS

Durante o desenvolvimento, descobrimos outro problema a ser tratado, os trechos de regras que contém \*(0 ou mais) ou Opt (0 ou 1?). Para tratá-los decidimos criar mais variáveis não terminais (e alterar as que forem necessárias), já que não achamos outra forma de conseguir fazer isso. Abaixo estão os problemas encontrados e suas respectivas soluções

```

stmt ->
|  if Exp then Block (elseif Exp then Block)* (else Block)opt end
|  return Expsopt
S2 ->  = Exp , Exp ( , Exp )opt do Block end in Exps do Block end Names in Exps do Block end

```

Nas do Stmt, foram feitas duas novas variáveis, SReturn e SIf, tendo as seguintes mudanças:

Stmt -> **if** Exp **Then** Block Sif **end** | **return** SReturn  
 Sif -> **elseif** Exp **then** Block Sif | **else** Block | Vazio  
 SReturn -> Exps | Vazio

S2 recebeu um tratamento semelhante, ficando assim:

S2 -> = Exp , Exp S2a **do** Block **end** | , Names **in** Exps **do** Block **end** | **in** Exps **do**  
 Block **end**

S2a -> , Exp | Vazio

$$\begin{array}{l} \textit{Exps} \\ ::= \textit{Exp} (, \textit{Exp})^* \end{array}$$

O Nao terminal Exps ficou da seguinte forma:

Exps -> Exp ExpsA  
 ExpsA -> , Exp ExpsA | Vazio

$$\begin{array}{l} \textit{Exp} \\ ::= \\ | \{ (\textit{Field} (, \textit{Field})^*)^{opt} \} \end{array}$$

Ja o Exp precisou de 2 novas variáveis para corrigir, sendo elas:

Exp -> { ExpA } E  
 ExpA -> Field ExpA2 | Vazio  
 ExpA2 -> , Field ExpA2 | Vazio

$$\begin{array}{l} \textit{Vars} \\ ::= \textit{Var} (, \textit{Var})^* \end{array}$$

Para Vars, foi criada uma variável no mesmo formato das repetições acima:

Vars -> Var V



V -> , Var V | Vazio

*FunctionBody*  
 $::= \text{Name} ( \text{Params}^{opt} ) \text{Block } \textbf{end}$

Na variável FunctionBody o resultado foi:

FunctionBody -> Name ( F ) Block **end**  
 F -> Param | Vazio

E por último, na variável Names, que está no formato Name (, Name)\*, que corrigida virou:

Names -> Name N  
 N-> , Name N | Vazio

O first e follow dessas novas variáveis, e das variáveis em que estes atributos mudaram estão presentes na tabela abaixo:

Variaveis	First	Follow
Sif	Else, Elseif, vazio	end
Sreturn	not, -, Name,(, function, {, nil, true, false, Number, String, vazio	;
S2a	virgula, vazio	do
ExpsA	virgula, vazio	; , do, virgula
ExpA	[, Name, vazio	}
ExpA2	virgula, vazio	}
F	Name, vazio	)
V	virgula, vazio	=
N	virgula, vazio	), in, =,

## 2. IMPLEMENTAÇÃO DO COMPILADOR

### 2.1. EXPLICAÇÃO DA BASE DO ALGORITMO

Agora, com essas tabelas e novas variáveis prontas, temos a base para começar a implementar o analisador sintático no nosso compilador. Primeiramente, tivemos de mudar a chamada do main, no programa. Antes era chamado direto na análise léxica, mas agora é feita apenas uma chamada para a análise sintática.

O corpo da chamada da análise sintática é o seguinte:

```
void SyntaxCheck() {  
  
    pilhaRegras[0] = EOF;  
    pilhaRegras[1] = 1;  
    // Starta a pilha com (pilha = Block EOF)  
  
    while (lastPilha >= 0) {  
  
        tokenSyntax = proximo_token();  
        while (tokenSyntax.nome_token != 0) {  
            switch (pilhaRegras[lastPilha])  
            {  
                case 1:  
                    Block();  
                    break;  
                case 2:  
                    Stmt();  
                    break;  
                case 3:  

```

Foram criadas duas variáveis no escopo global, por fins de praticidade, que seriam a pilhaRegras[] e tokenSyntax. A variável tokenSyntax é o token que está sendo recebido da análise Léxica, a mando do bloco Sintático. Já PilhaRegras, é onde são empilhadas as regras da gramática, que foram numeradas de 1 a 27 (Block = 1, Stmt = 2 [...]), assim deixando a estrutura do algoritmo semelhante ao exemplo visto no slide, mas com os tokens da cadeia lidos um a um.

Pilha	Cadeia
E\$	id v id & id \$

No final do bloco, existe o case EOF, que é quando o topo da pilha contém o símbolo indicativo de fim. Se o token recebido da análise também for EOF a cadeia é reconhecida e a análise é encerrada. Se o token ainda não for EOF, o topo da pilha aumenta, e é posto o 1 (Block) que é a variável inicial da regra, para que se possa continuar com a leitura até que se chegue no fim do código.

O default – que é ativado quando nenhum dos Case é correspondido, ou seja, há um Token no topo da pilha – serve para duas coisas, ativar a função de consumir o token quando a entrada for igual ao que está no topo da pilha, e caso sejam diferentes, o código trata esse erro devidamente.

```
case EOF:
    if (pilhaRegras[lastPilha] == tokenSyntax.nome_token) {
        printf("Cadeia reconhecida.");
        return;
    }
    else {
        lastPilha += 1;
        pilhaRegras[lastPilha] = 1;
        break;
    }

default:
    if (tokenSyntax.nome_token == pilhaRegras[lastPilha]) {
        printf("Consome()\n\n");
        Consumir();
        break;
    }
    else {
        TrataErro();
        break;
    }
};
```

Já as regras da gramática estão todas no seguinte formato, elas comparam o token recebido da análise léxica com aquelas presentes em seu conjunto First. Caso o token faça parte do conjunto, é executada a ação presente na tabela preditiva (Ela empilha a regra que a leva até esse Terminal)

```
void PrefixExp() { //9
    switch (tokenSyntax.nome_token)
    {
        case ID: // Name P
            pilhaRegras[lastPilha] = 10; // P
            lastPilha += 1;
            pilhaRegras[lastPilha] = ID;
            break;

        case OPR: // ( Exp ) P
            pilhaRegras[lastPilha] = 10; // P
            lastPilha += 1;
            pilhaRegras[lastPilha] = CPR;
            lastPilha += 1;
            pilhaRegras[lastPilha] = 7; // Exp
            lastPilha += 1;
            pilhaRegras[lastPilha] = OPR;
            break;
    }
}
```

Quando essa regra possui vazio como parte do conjunto First, são incluídos os elementos do conjunto Follow, para que a análise não resulte em erro quando encontrar esses tokens (entrar no case Default), apenas diminuindo em 1 o topo da pilha, para tirar a leitura da sua própria regra e continuar a análise normalmente.

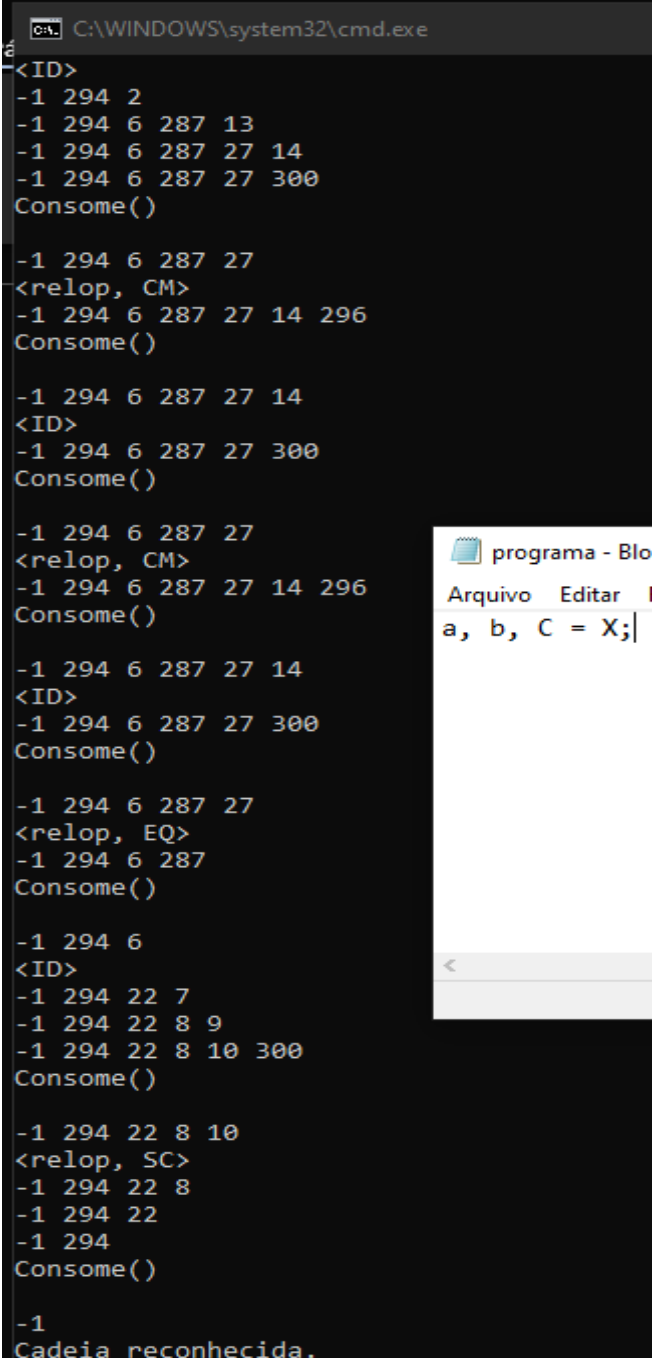
```
case SC: // Follow
    lastPilha -= 1;
    break;

case CBK:
    lastPilha -= 1;
    break;
```

### 2.1.1. TESTES FEITOS

Os testes foram feitos escrevendo algumas cadeias que devem ser aceitas pela linguagem. Selecionamos algumas para colocar abaixo:

(No código, foram colocados alguns prints para mostrar o processo de movimentação da pilha, o token que está causando essa movimentação, e a parte do consumo de token para ficar mais compreensível.)



```
C:\WINDOWS\system32\cmd.exe
<ID>
-1 294 2
-1 294 6 287 13
-1 294 6 287 27 14
-1 294 6 287 27 300
Consome()

-1 294 6 287 27
<relop, CM>
-1 294 6 287 27 14 296
Consome()

-1 294 6 287 27 14
<ID>
-1 294 6 287 27 300
Consome()

-1 294 6 287 27
<relop, CM>
-1 294 6 287 27 14 296
Consome()

-1 294 6 287 27 14
<ID>
-1 294 6 287 27 300
Consome()

-1 294 6 287 27
<relop, EQ>
-1 294 6 287
Consome()

-1 294 6
<ID>
-1 294 22 7
-1 294 22 8 9
-1 294 22 8 10 300
Consome()

-1 294 22 8 10
<relop, SC>
-1 294 22 8
-1 294 22
-1 294
Consome()

-1
Cadeia reconhecida.
```

```
C:\WINDOWS\system32\cmd.exe
Consome()
-1 294 261 19
<ELSE>
-1 294 261 1 263
Consome()
-1 294 261 1
<RETURN>
-1 294 261 294 2
-1 294 261 294 20 260
Consome()
-1 294 261 294 20
<ID>
-1 294 261 294 6
-1 294 261 294 22 7
-1 294 261 294 22 8 9
-1 294 261 294 22 8 10 300
Consome()
-1 294 261 294 22 8 10
<relop, SC>
-1 294 261 294 22 8
-1 294 261 294 22
-1 294 261 294
Consome()
-1 294 261
<END>
Consome()
-1 294
<relop, SC>
Consome()
-1
Cadeia reconhecida.
```

programa - Bloco de Notas

Arquivo	Editar	Formatar	Ex
---------	--------	----------	----

```
if (a<7) then
    a = 10;
elseif (a>7) then
    a = 4;
else
    return a;
end;
```

```
C:\WINDOWS\system32\cmd.exe

-1 294 261 294 261 294 261 294 22 7
<NUM>
-1 294 261 294 261 294 261 294 22 8 301
Consome()

-1 294 261 294 261 294 261 294 22 8
<relop, SC>
-1 294 261 294 261 294 261 294 22
-1 294 261 294 261 294 261 294
Consome()

-1 294 261 294 261 294 261
<END>
Consome()

-1 294 261 294 261 294
<relop, SC>
Consome()

-1 294 261 294 261
<END>
Consome()

-1 294 261 294
<relop, SC>
Consome()

-1 294 261
<END>
Consome()

-1 294
<relop, SC>
Consome()

-1
Cadeia reconhecida.
```

```
programa - Bloco de Notas
Arquivo  Editar  Formatar  Exibir  Ajuda
local function TestaSintaxe (a, b, c)
    for teste = 0, teste<=10, teste + 2 do
        if ((teste - 5) > 0) then
            a = teste;
        else
            a = a * 2;
        end;
    end;
end;
```

### 3. TRATAMENTO DE ERROS

Em relação ao tratamento de erros, não posso dizer que conseguimos o implementar com sucesso. A base funciona, mas dependendo da cadeia ocorrem muitos problemas, como os clássicos erros em cascatas e até mesmo loops causados por forças desconhecidas (possivelmente coisa da linguagem, pois até tentamos fazer gambiarra para forçar o programa a sair do Bloco problemático para testes, e mesmo assim o loop continuava, e acontecia até mesmo sem iterações / chamada recursiva.)

Mas, até certo ponto ele é funcional, então apresentarei a lógica que implementamos.

```
else if ((tokenSyntax.nome_token == EOF) && contaErro > 0) {  
    printf("Cadeia rejeitada.\n %d erro(s) encontrado(s).", contaErro);  
    return;  
}
```

Agora, uma outra forma de encerrar o programa é essa, aonde o token carregado é o próprio EOF, então é imprimido a quantidade de erros encontrados e retorna a função. (Quando ocorre erros em cascata, obviamente esse valor acaba por estar errado.)

Quando um bloco lê um terminal em que ele não possua nenhuma correspondência (Não está presente no first nem follow (sync)), ele chama a função NeedSync, que printa o terminal / Regra esperada e o que foi recebido, e seta uma nova variável Bool Sinc para false, ou seja, diz que o programa não está mais sincronizado. (O token ainda não é descartado pois isso acontece no início da próxima função)

```
default:  
    NeedSync();  
};
```

```
void NeedSync() {  
    printf("Erro Sintatico.\nEsperava ");  
    EmitirMensagem(); // Emite o esperado  
    printf("Antes de: ");  
    TokenRecebido(); // Diz o token recebido da analise lexica  
    sinc = 0;  
}
```



Quando sinc é false, ao sair do Switch Case principal, que leva o token até a regra que está na pilha, ele cai numa função que varre a entrada até achar um token sincronizável na entrada, ou seja, enquanto não houver correspondência do token com a regra que está no topo da pilha, esse token é descartado e é lido um próximo até que se possa resumir a análise. Nesse ponto, é aumentado o contador de erros, que já impossibilita a cadeia de ser aceita (Se existe mais de 0 erros, a análise está continuando apenas para avaliar o resto do programa).

```
if (sinc == 0) { //Varre a entrada ate achar uma variavel sincronizavel
    CheckSinc();
}
```

```
3171 void CheckSinc() {
3172     contaErro += 1;
3173     while (sinc == 0) {
3174         tokenSyntax = proximo_token();
3175         printf("%d\n", tokenSyntax.nome_token);
3176         switch (pilhaRegras[lastPilha])
3177         {
3178             case 1:
3179                 Block();
3180                 break;
3181             case 2:
3182                 Stmt();
3183                 break;
```

Ao achar um token sincronizável (O Follow de cada regra possui um case que o leva para a função Sync() ), que retorna a variável sinc para True, e remove o topo da pilha.

```
//tokens sync
case ELSEIF:
    Sync();
    break;

case ELSE:
    Sync();
    break;
```

```

1435 void Sync() {
1436     printf("Erro Sintatico.\nEsperava ");
1437     EmitirMensagem(); // Emite o esperado
1438     printf("Antes de: ");
1439     TokenRecebido(); // Diz o token recebido da analise lexica
1440     contaErro += 1;
1441     lastPilha -= 1;
1442     sinc = 1;
1443 }

```

E por último, no caso em que existe um não terminal no topo da pilha e ele não coincide com o token recebido da análise léxica, o código entra em uma função, onde é informado o token recebido e o que era esperado, e remove o topo da pilha. Caso o próximo elemento da pilha seja um terminal, ele continua fazendo a verificação (Se o token lido e o topo são iguais) e removendo no caso de não serem iguais.

```

3401
3402
3403
3404
3405
3406
3407
3408
3409
3410
3411
3412
3413
3414

```

```

default:
    if (tokenSyntax.nome_token == pilhaRegras[lastPilha]) {
        printf("Consome");
        TokenRecebido();
        printf("\n");
        Consumir();
        break;
    }
    else {
        //printf("Entro tratamento");
        TrataErro();
        break;
    }
};

```

```

1454 void TrataErro() {
1455     printf("Erro Sintatico.\n");
1456     printf("Esperava: ");
1457     EmitirMensagem(); //Emite token esperado
1458     printf("Foi recebido: ");
1459     TokenRecebido(); // Diz o token recebido da analise lexica
1460     printf("\n");
1461     lastPilha -= 1;
1462     contaErro += 1;
1463
1464 }

```

### 3.1. TESTES DE RECUPERAÇÃO DE ERROS

Exemplo de testes sucedidos:

Problema no txt – “[...] teste - ) [...]”

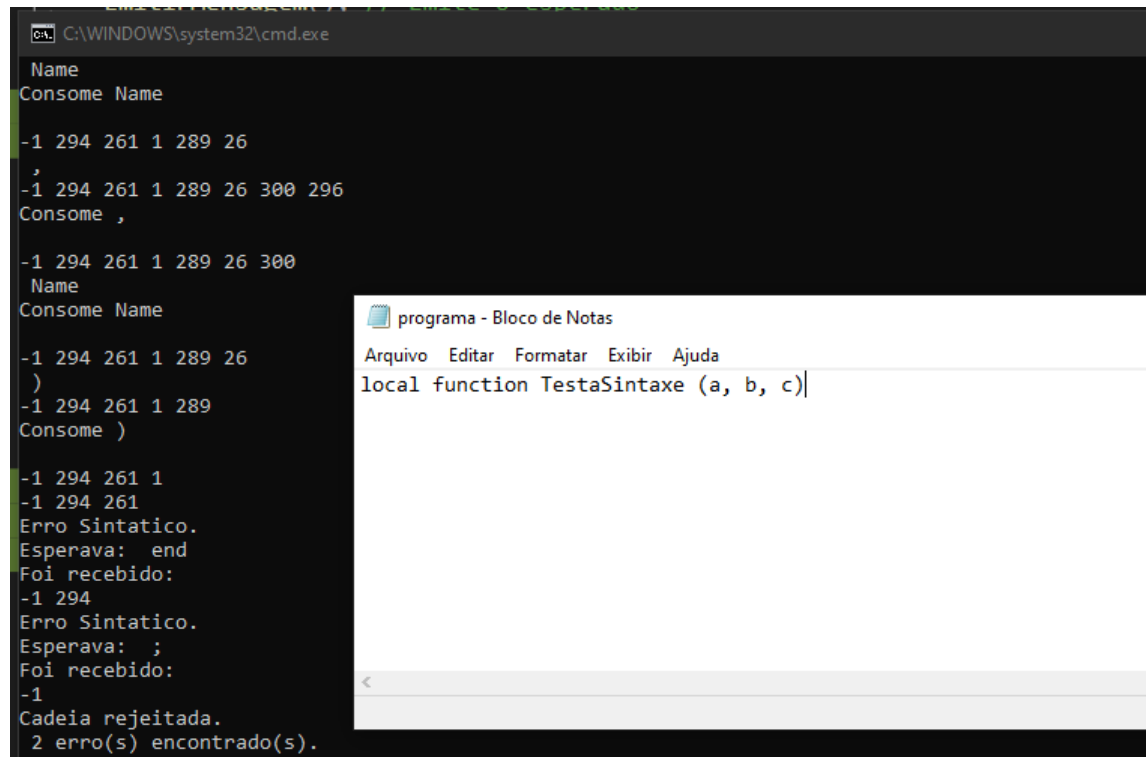
```
C:\WINDOWS\system32\cmd.exe
-1 294 261 294 261 19 1 265 8 10 289 8 9
-1 294 261 294 261 19 1 265 8 10 289 8 10 289 7 288
Consome (
-1 294 261 294 261 19 1 265 8 10 289 8 10 289 7
Name
-1 294 261 294 261 19 1 265 8 10 289 8 10 289 8 9
-1 294 261 294 261 19 1 265 8 10 289 8 10 289 8 10 300
Consome Name
-1 294 261 294 261 19 1 265 8 10 289 8 10 289 8 10
-
-1 294 261 294 261 19 1 265 8 10 289 8 10 289 8
-1 294 261 294 261 19 1 265 8 10 289 8 10 289 7 12
-1 294 261 294 261 19 1 265 8 10 289 8 10 289 7 277
Consome -
-1 294 261 294 261 19 1 265 8 10 289 8 10 289 7
)
Erro Sintatico.
Esperava First(Exp)
Antes de: )
-1 294 261 294 261 19 1 265 8 10 289 8 10 289
Consome )
-1 294 261 294 261 19 1 265 8 10 289 8 10
>
-1 294 261 294 261 19 1 265 8 10 289 8
-1 294 261 294 261 19 1 265 8 10 289 7 12
-1 294 261 294 261 19 1 265 8 10 289 7 286
```

```
programa - Bloco de Notas
Arquivo Editar Formatar Exibir Ajuda
for teste = 0, teste<=10, teste + 2 do
  if ((teste - ) > 0) then
    a = teste;
  end;
end; |
```

```
-1 294
;
Consome ;

-1
Cadeia rejeitada.
1 erro(s) encontrado(s).
```

Problemas no txt – Falta end; no final



The image shows a Windows command prompt window and a Notepad window. The command prompt window displays the output of a program, which includes several lines of data and error messages. The Notepad window shows a JavaScript function definition with a syntax error.

```
C:\WINDOWS\system32\cmd.exe
Name
Consome Name
-1 294 261 1 289 26
,
-1 294 261 1 289 26 300 296
Consome ,
-1 294 261 1 289 26 300
Name
Consome Name
-1 294 261 1 289 26
)
-1 294 261 1 289
Consome )
-1 294 261 1
-1 294 261
Erro Sintatico.
Esperava: end
Foi recebido:
-1 294
Erro Sintatico.
Esperava: ;
Foi recebido:
-1
Cadeia rejeitada.
2 erro(s) encontrado(s).
```

programa - Bloco de Notas

Arquivo Editar Formatar Exibir Ajuda

```
local function TestaSintaxe (a, b, c)|
```

Exemplo de erro que gera erros em cadeia: Semelhante ao erro acima, mas com um adicional de [...] teste - ) 0)[...], tendo 2 erros no total, mas contando 9 no programa.

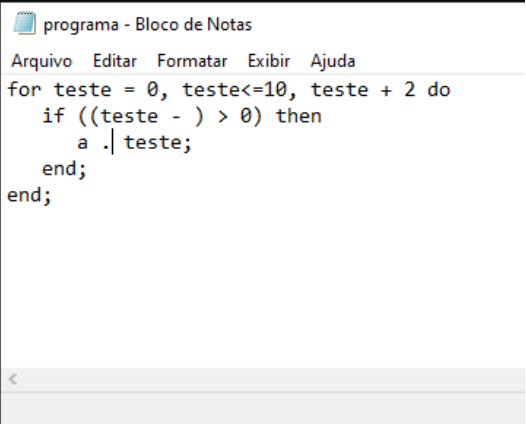
```
C:\WINDOWS\system32\cmd.exe
-
-1 294 261 294 261 19 1 265 8 10 289 8 10 289 8
-1 294 261 294 261 19 1 265 8 10 289 8 10 289 7 12
-1 294 261 294 261 19 1 265 8 10 289 8 10 289 7 277
Consome -
-1 294 261 294 261 19 1 265 8 10 289 8 10 289 7
)
Erro Sintatico.
Esperava First(Exp)
Antes de: )
-1 294 261 294 261 19 1 265 8 10 289 8 10 289
Consome )
-1 294 261 294 261 19 1 265 8 10 289 8 10
Number
Erro Sintatico.
Esperava First(P)
Antes de: Number
289
-1 294 261 294 261 19 1 265 8 10 289 8
265
-1 294 261 294 261 19 1 265 8 10 289
300
-1 294 261 294 261 19 1 265 8 10
-1 294 261 294 261 19 1 265 8 10
Erro Sintatico.
Esperava First(P)
Antes de: Name
287
```

```
programa - Bloco de Notas
Arquivo Editar Formatar Exibir Ajuda
for teste = 0, teste<=10, teste + 2 do
  if ((teste - ) 0) then
    a = teste;
  end;
end;
```

```
-1 294 261
Erro Sintatico.
Esperava: end
Foi recebido:
-1 294
Erro Sintatico.
Esperava: ;
Foi recebido:
-1
Cadeia rejeitada.
9 erro(s) encontrado(s).
```

Exemplo de erro que leva a um loop: Igual ao erro acima, mas com A . teste;, aonde se esperaria um A = teste.

```
-1
Erro Sintatico.
Esperava First(V)
Antes de: -1 294 261 294 261 19 294 6 287 27
-1
Erro Sintatico.
Esperava First(V)
Antes de: -1 294 261 294 261 19 294 6 287 27
-1
Erro Sintatico.
Esperava First(V)
Antes de: -1 294 261 294 261 19 294 6 287 27
-1
Erro Sintatico.
Esperava First(V)
Antes de: -1 294 261 294 261 19 294 6 287 27
-1
Erro Sintatico.
Esperava First(V)
Antes de: -1 294 261 294 261 19 294 6 287 27
-1
Erro Sintatico.
Esperava First(V)
```



```
programa - Bloco de Notas
Arquivo  Editar  Formatar  Exibir  Ajuda
for teste = 0, teste<=10, teste + 2 do
  if ((teste - ) > 0) then
    a .| teste;
  end;
end;
```

OBS: Pedimos desculpas pelos bugs nessa parte. Tentamos muito resolver, mas não saímos disso, fora que muitas vezes até piorava. Não temos certeza se é em relação às regras que fizemos, ou se o algoritmo tem uma falha que não enxergamos... Estamos enviando 2 algoritmos, um com o tratamento, e outro antes de aplicarmos o tratamento, já que não sabemos se ele pode ter resultado em alguma falha em cadeias que antes seriam aceitas (Em todas as que testamos, aparente estar OK). Enfim, obrigado pela compreensão!