Etapa 2: Análise Sintática (Grupo 3)

Nomes: Rômulo Peigas Fonseca e Leonardo Bacellar

1. Verifique se a gramática da linguagem LS, descrita acima, é LL(1) e, se necessário, transforme-a.

Na primeira análise da gramática identificamos que *Stmt* precisava ser fatorada a esquerda, que a recursão a esquerda de *Exp* e que a recursão indireta de PrefixExp e Var precisavam ser removidas.
Após fazermos as devidas alterações, começamos a escrever o pseudo-código e percebemos que a gramática precisava de mais alterações.

O mesmo método para remover recursão indireta aplicado em PrefixExp, também precisou ser aplicado em Var e Stmt_1 (que foi gerado após a primeira fatoração de Stmt), pois sem isso haveria pontos de indecisão na gramática.
Removemos Name da regra *Stmt_2* ::= **function** Name *FunctionBody*, pois continuando a derivação Name se repete gerando a redundância a seguir:
function Name Name **(** *Params*^{opt} **)** *Block* **end**

Em Stmt tiramos o “opt” do local, na regra:

Stmt ::= **local**^{opt} **function** *FunctionBody*

para facilitar a segunda fatoração, pois se local não existir o próximo símbolo é **function**, o que significa que no momento da verificação o analisador vai seguir para a produção *Stmt* ::= *Function*.
Feitas as modificações listadas acima obtivemos a gramática abaixo.

Block ::= (*Stmt*;)*

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** Exps^{opt}
| **break**
| **for** *Stmt_1*
| **local** *Stmt_2*

Stmt_1
::= Name *Stmt_11*

Stmt_11
::= = *Exp*, *Exp* (,*Exp*)^{opt} **do** *Block* **end**
| (, Name)* **in** *Exps* **do** *Block* **end**

Stmt_2
::= **function** *FunctionBody*
| *Names* = *Exps*

Exps ::= *Exp* (, *Exp*)*

Exp
::= **not** *Exp Exp_1*
| **-** *Exp Exp_1*
| *PrefixExp Exp_1*
| *Function Exp_1*
| **{** (*Field* (, *Field*)*)^{opt} **}** *Exp_1*
| **nil** *Exp_1*
| **true** *Exp_1*
| **false** *Exp_1*
| Number *Exp_1*
| String *Exp_1*

Exp_1
::= *BinOp Exp Exp_1*
| **ε**

PrefixExp

::= Name *PrefixExp_1*
| **(** *Exp* **)** *PrefixExp_1*

PrefixExp_1
::= **[** *Exp* **]** *PrefixExp_1*
| **ε**

Field
::= **[** *Exp* **]** = *Exp*
| Name = *Exp*

BinOp
::= **or** | **and** | **<** | **>** | **<=** | **>=** | **~=** | **==** | **..** | **+** | **-** | ***** | **/** | **^**

Vars ::= *Var* (**,** *Var*)*

Var
::= Name *Var_1*
| **(** *Exp* **)** *PrefixExp_1* **[** *Exp* **]**

Var_1
::= *PrefixExp_1* **[** *Exp* **]**
| ε

Function ::= **function** *FunctionBody*

FunctionBody ::= Name **(** *Params*^{opt} **)** *Block* **end**

Params ::= *Names*

Names ::= Name (**,** Name)*

2. Construa os procedimentos recursivos preditivos para a linguagem LS baseado em sua gramática.

Para fazer os procedimentos preditivos recursivos calculamos os conjuntos First e Follow de cada não terminal, que se encontram abaixo e as funções do analisador sintático se encontram no código em anexo. Como o pseudo-código do analisador ficou muito extenso, não foi possível colocá-lo neste relatório.

First(*Block*) = First(*Stmt*) = {Name, (, function, do, while, if, return, break, for, local}
First(*Stmt_1*) = {Name}
First(*Stmt_11*) = { = , **“,”**, in}
First(*Stmt_2*) = {function, Name}
First(*Exps*) = First(*Exp*) = {not, -, Name,(,function,{,nil,true,false,Number, String}
First(*Exp_1*) = {or, and, <, >, <=, >=, ~=, ==, .., +, -, *, /, ^, ε}
First(*PrefixExp*) = {Name, (}
First(*PrefixExp_1*) = { [, ε}
First(*Field*) = { [, Name}
First(*BinOP*) = {or, and, <, >, <=, >=, ~=, ==, .., +, -, *, /, ^}
First(*Vars*) = First(*Var*) = {Name, (}
First(*Var_1*) = { [, **ε**}
First(*Function*) = {function}
First(*FunctionBody*) = {Name}
First(*Params*) = First(*Names*) = {Name}

Follow(*Block*) = {$,end,elseif,else}
Follow(*Stmt*) = Follow(*Stmt_1*) = Follow(*Stmt_2*) = Follow(*Stmt_11*) = {**;**}
Follow(*Exps*) = {**;**, do}
Follow(*Exp*) = Follow(*Exp_1*) = {do,then, **","** , **";"**,],), } }
Follow(*PrefixExp*) = Follow(*PrefixExp_1*) = { [, do,then, **","** , **";"**,],), } }
Follow(*Field*) = { }, **“,”** }
Follow(*BinOP*) = {not, -, Name,(,function,{,nil,true,false,Number, String}
Follow(*Vars*) = { = }
Follow(*Var*) = Follow(*Var_1*) = {=, **,**}
Follow(*Function*) = Follow(*FunctionBody*) = {do,then, **","** , **";"**,],), } }

Follow(*Params*) = {) }
Follow(*Names*) = {) , = }

3. Altere a chamada no programa principal (do Trabalho 1). Agora quem comandará a análise é o analisador sintático. A cada token retornado (procedimento ObterToken do analisador léxico) deverá ser acionado o procedimento que o analisa.

Feito no código fonte em anexo.

4. Implementar o tratamento de erro usando o modo pânico, sempre relatando os erros para o usuário. Determinar os símbolos de sincronização possíveis em cada caso.

O tratamento de erro foi implementado no código e os conjuntos de sincronização seguem abaixo.

Sinc(*Block*) = {$,end,elseif,else, **;** , do,then, **","** , **";"**,] ,) , } }
Sinc(*Stmt*) = {**;** , $,end,elseif,else}
Sinc(*Stmt_1*) = {**;**}
Sinc(*Stmt_2*) = {**;**}
Sinc(*Stmt_11*) = {**;**}
Sinc(*Exps*) = {**;** , do}
Sinc(*Exp*) = {do,then, **","** , **";"**,) , } , , [,] , = }
Sinc(*Exp_1*) = {do,then, **","** , **";"**,],), } }
Sinc(*PrefixExp*) = { [, do,then, **","** , **";"**,],), } }
Sinc(*PrefixExp_1*) = { [, do,then, **","** , **";"**,],), } , =}
Sinc(*Field*) = { }, **“,”** do,then , **";"**,],),}
Sinc(*BinOP*) = {not, -, Name,(,function,{,nil,true,false,Number, String, do,then, **","** , **";"**,],), }}
Sinc(*Vars*) = { = , **;**}
Sinc(*Var*) = { =, **‘,’** }
Sinc(*Var_1*) = { =, **‘,’** }
Sinc(*Function*) = {do,then, **","** , **";"**,],), } }
Sinc(*FunctionBody*) = { do,then, **","** , **";"**,],), } }
Sinc(*Params*) = {) , do,then, **","** , **";"**,], }}
Sinc(*Names*) = {) ,= }

5. Relate detalhadamente o funcionamento do analisador sintático construído, incluindo no relatório: descrição teórica do programa; descrição da sua estrutura; descrição de seu funcionamento; descrição dos testes realizados e das saídas obtidas.

Primeiro transformamos todas as não terminais da gramática em funções, para podermos fazer as chamadas recursivas simulando uma estrutura de pilha.
A função Block, que representa o símbolo inicial, pede um token e verifica se este pertence ao First(Stmt), se pertencer é feita uma chamada da função Stmt, que dentro dela verifica o nome do token ou seu atributo, quando necessário, e verifica a qual produção ele pertence. Dessa forma o programa continua chamando os tokens conforme a necessidade, verificando em que produção ele se encaixa e chamando as funções correspondentes aos não terminais que o aceitam.
Caso o token não corresponda ao esperado, o programa tenta recuperar o erro chamando uma função erro, que vai verificar se este token errado faz parte do conjunto de sincronização e continuará procurando por um token que faça parte deste conjunto, para depois dar continuidade a leitura.
Seguem nas próximas páginas os prints dos testes.

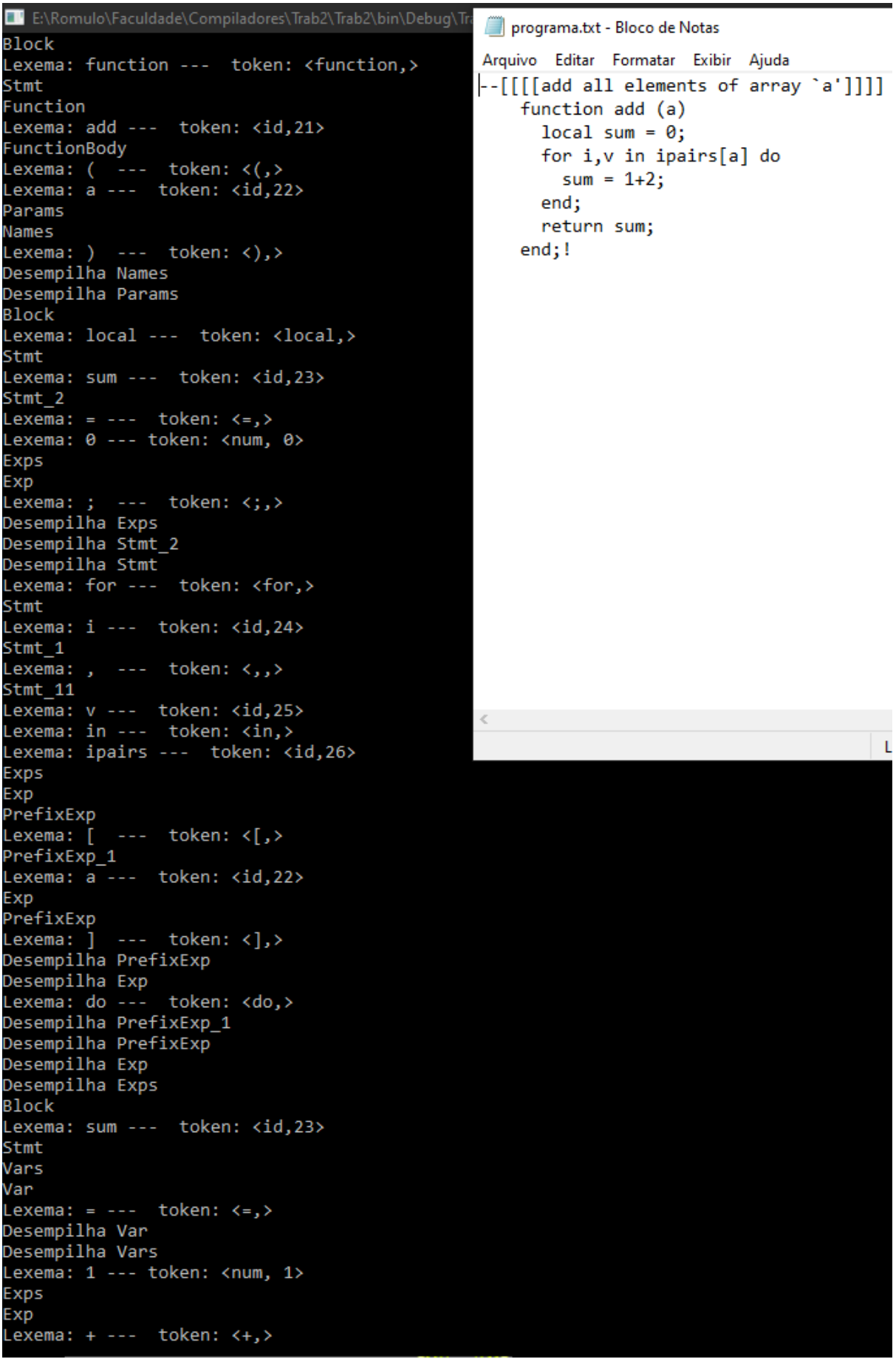
```
Block
Lexema: string1 ---  token: <id,21>
Stmt
Vars
Var
Lexema: = ---  token: <=,>
Desempilha Var
Desempilha Vars
Lexema: "Lua" --- token: <string, 22>
Exps
Exp
Lexema: ;  ---  token: <;,>
Desempilha Exps
Desempilha Stmt
Lexema: string2 ---  token: <id,23>
Stmt
Vars
Var
Lexema: = ---  token: <=,>
Desempilha Var
Desempilha Vars
Lexema: "Tutorial" --- token: <string, 24>
Exps
Exp
Lexema: ;  ---  token: <;,>
Desempilha Exps
Desempilha Stmt
Lexema: string_concat ---  token: <id,25>
Stmt
Vars
Var
Lexema: = ---  token: <=,>
Desempilha Var
Desempilha Vars
Lexema: string1 ---  token: <id,21>
Exps
Exp
PrefixExp
Lexema: .. ---  token: <del_op,..>
Desempilha PrefixExp
Exp_1
Binop
Desempilha Binop
Lexema: string2 ---  token: <id,23>
Exp
PrefixExp
Lexema: ;  ---  token: <;,>
Desempilha PrefixExp
Desempilha Exp
Desempilha Exp_1
Desempilha Exp
Desempilha Exps
Desempilha Stmt
Lexema: ! ---  token: <!,>
Desempilha Block
```

programa.txt - Bloco de Notas

Arquivo Editar Formatar Exibir Ajuda

```
string1 = "Lua";
string2 = "Tutorial";

--[[Concatenando as strings 1 e 2
usando o operador ..]]
string_concat = string1..string2;
!
```


E:\Romulo\Faculdade\Compiladores\Trab2\Trab2\bin\Debug\Tr
Block
Lexema: function --- token: <function,>
Stmt
Function
Lexema: add --- token: <id,21>
FunctionBody
Lexema: (--- token: <(,>
Lexema: a --- token: <id,22>
Params
Names
Lexema:) --- token: <),>
Desempilha Names
Desempilha Params
Block
Lexema: local --- token: <local,>
Stmt
Lexema: sum --- token: <id,23>
Stmt_2
Lexema: = --- token: <=,>
Lexema: 0 --- token: <num, 0>
Exps
Exp
Lexema: ; --- token: <;,>
Desempilha Exps
Desempilha Stmt_2
Desempilha Stmt
Lexema: for --- token: <for,>
Stmt
Lexema: i --- token: <id,24>
Stmt_1
Lexema: , --- token: <,,>
Stmt_11
Lexema: v --- token: <id,25>
Lexema: in --- token: <in,>
Lexema: ipairs --- token: <id,26>
Exps
Exp
PrefixExp
Lexema: [--- token: <[,>
PrefixExp_1
Lexema: a --- token: <id,22>
Exp
PrefixExp
Lexema:] --- token: <],>
Desempilha PrefixExp
Desempilha Exp
Lexema: do --- token: <do,>
Desempilha PrefixExp_1
Desempilha PrefixExp
Desempilha Exp
Desempilha Exps
Block
Lexema: sum --- token: <id,23>
Stmt
Vars
Var
Lexema: = --- token: <=,>
Desempilha Var
Desempilha Vars
Lexema: 1 --- token: <num, 1>
Exps
Exp
Lexema: + --- token: <+,>
programa.txt - Bloco de Notas
Arquivo Editar Formatar Exibir Ajuda
--[[[[add all elements of array `a']]]]
function add (a)
local sum = 0;
for i,v in ipairs[a] do
sum = 1+2;
end;
return sum;
end;!

```
Exps
Exp
Lexema: + ---  token: <+,>
Exp_1
Binop
Desempilha Binop
Lexema: 2 --- token: <num, 2>
Exp
Lexema: ;  ---  token: <;,>
Desempilha Exp_1
Desempilha Exp
Desempilha Exps
Desempilha Stmt
Lexema: end ---  token: <end,>
Desempilha Block
Lexema: ;  ---  token: <;,>
Desempilha Stmt_11
Desempilha Stmt_1
Desempilha Stmt
Lexema: return ---  token: <return,>
Stmt
Lexema: sum ---  token: <id,23>
Exps
Exp
PrefixExp
Lexema: ;  ---  token: <;,>
Desempilha PrefixExp
Desempilha Exp
Desempilha Exps
Desempilha Stmt
Lexema: end ---  token: <end,>
Desempilha Block
Lexema: ;  ---  token: <;,>
Desempilha FunctionBody
Desempilha Function
Desempilha Stmt
Lexema: ! ---  token: <!,>
Desempilha Block
```