

# Trabalho Prático 1 - Parte 2: Análise Sintática

Geovanna Menegasse Silva  
Vitor Lucio dos Santos Ferreira

## 1. Introdução

Esse trabalho consiste no desenvolvimento de um analisador sintático para a linguagem TIGER, segunda etapa para o desenvolvimento de um compilador completo. Essa fase é responsável por pegar os tokens identificados no analisador léxico e analisar se a sintaxe do programa está correta, usando uma Gramática Livre de Contexto (GLC).

Nas próximas seções apresentaremos a descrição da solução encontrada e as instruções de utilização do analisador sintático criado. Além disso, apresentaremos alguns dos resultados obtidos pela submissão dos códigos fonte feitos na linguagem TIGER, disponibilizados para teste, ao nosso analisador sintático.

## 2. Descrição da solução

A solução para este trabalho prático foi, inicialmente, definir as regras de associatividade e precedência dos tokens usados no analisador léxico e, em seguida, escrever a mesma GLC, encontrada no pdf da linguagem TIGER, no código do analisador sintático.

Tivemos conflitos *shift-reduce* que solucionamos adicionando as regras de precedência e associatividade mencionadas anteriormente. Também tivemos um conflito *reduce-reduce* durante a criação do analisador sintático e iremos explicar a resolução dele em um tópico mais adiante.

Para a criação de um executável para nosso analisador sintático, foi feita a transformação do “*scanner.l*” no arquivo “*lex.yy.c*”, para a parte do analisador léxico e a transformação do arquivo “*sintaxe.y*” nos arquivos “*y.tab.h*”, que contém a declaração dos tokens para serem usados no scanner, e o “*y.tab.c*”, que contém a implementação a ser usada para criar o analisador sintático, contendo os estados e transições LALR(1) correspondentes a GLC usada. Ao final, criou-se o executável “*a.out*” que representa o analisador sintático, o qual recebe um código fonte TIGER como entrada e responde se a sintaxe está correta ou não.

O debug do analisador sintático foi feito usando o arquivo “y.output”, que também foi gerado a partir da transformação do arquivo “sintaxe.y”, contendo uma versão mais legível dos estados e transições.

Para as transformações mencionadas e criação do analisador sintático, utilizamos os comandos:

```
flex scanner.l
yacc -v -d sintaxe.y
cc y.tab.c
```

Por fim, para facilitar a compilação e execução do programa, criamos um arquivo makefile que executa esses comandos.

### 3. Conflitos *shift-reduce* e *reduce-reduce*

**Conflito *shift-reduce*:** conflitos desse tipo foram identificados pelo yacc devido a ambiguidades na gramática causadas pela falta de definição de precedência e associatividade nas expressões aritméticas, nas estruturas condicionais *if-then-else* e nas estruturas de repetição *while-do*. Como dito anteriormente, para solucionar estes conflitos, definimos regras de precedência e associatividade para os terminais conflitantes, forçando o yacc a escolher sempre a opção do *shift* nesses casos.

**Conflito *reduce-reduce*:** o yacc identificou uma ambiguidade na gramática referente aos não-terminais *l\_value* e *type\_id*, junto com o terminal *id*, gerando um conflito *reduce-reduce*. Para solucionar este conflito, duplicamos todas as produções referentes ao *l\_value*, colocamos *type\_id* no lugar de *l\_value* nestas duplicações e removemos a regra onde *l\_value* poderia virar o terminal *id*.

### 4. Gramática modificada

```
exp: type-id | l-value | nil | "(" expseq ")" | num | string
    | - exp
    | id "(" args ")"
    | exp "+" exp | exp "-" exp | exp "*" exp | exp "/" exp
    | exp "=" exp | exp "<>" exp
    | exp "<" exp | exp ">" exp | exp "<=" exp | exp ">=" exp
    | exp "&" exp | exp "|" exp
    | type-id "{" id "=" exp idexps "}"
    | type-id "[" exp "]" of exp
    | l-value ":" exp
    | type-id ":" exp
    | if exp then exp else exp
```

```

| if exp then exp
| while exp do exp
| for id ":" exp to exp do exp
| break
| let decs in expseq end

decs: dec decs | ε
dec: tydec | vardec | fundec
tydec: type id "=" ty
ty: id | "{" id ":" type-id tyfields1 "}" | array of id
tyfields: id ":" type-id tyfields1 | ε
tyfields1: "," id ":" type-id tyfields1 | ε
vardec: var id ":" exp | var id ":" type-id ":" exp
fundec: function id "(" tyfields )" " = " exp
      | function id "(" tyfields )" ":" type-id "=" exp

l-value: l-value ." id | l-value "[" exp "]"
        | type-id ." id | type-id "[" exp "]"
type-id: id

expseq: exp expseq1 | ε
expseq1: ;" exp expseq1 | ε

args: exp args1 | ε
args1: "," exp args1 | ε

idexps: "," id "=" exp idexps | ε

```

## 5. Exemplos de teste

Para testarmos nosso analisador sintático, submetemos ao programa os códigos disponibilizados para teste. Apresentamos aqui apenas dois testes dentre os utilizados: o primeiro com a sintaxe correta e o segundo com erro de sintaxe. Utilizando os comandos `./a.out tests/08.txt` e `./a.out tests/06.txt` submetemo-nos para análise.

### 5.1. Exemplo com sintaxe correta

O resultado a seguir é obtido ao submeter o arquivo de teste número oito ao nosso analisador sintático. A saída do programa contém a lista de regras utilizadas para acompanhar o reconhecimento sintático, o resultado da análise sintática e a listagem do código fonte submetido ao analisador.

O acompanhamento do analisador sintático pelas regras e pelo código fonte, é feito de baixo para cima. Por exemplo: a primeira regra detectada no programa abaixo foi a regra “*exp -> let decs in expseq end*”. Partindo dessa regra, o reconhecedor sintático escolheu o símbolo não-terminal mais à direita para traduzir, que no nosso caso é o símbolo “*expseq*”. Esse símbolo é reconhecido na expressão “ *Arranjo[2]* = “*nome*” ” no nosso programa e, por isso, o analisador o traduziu, depois de uma sequência de traduções, para a produção “*exp -> exp = exp*”, onde o primeiro *exp* é um *id [ num ]* e o segundo *exp* é um *string*. Só após terminar a tradução deste símbolo, o analisador passa para o próximo não-terminal “*decs*” para traduzi-lo.

#### REGRAS UTILIZADAS:

```
ty -> array of id
tydec -> type id = ty
dec -> tydec
type-id -> id
type-id -> id
exp -> num
exp -> num
exp -> type-id [ exp ] of exp
vardec -> var id : type-id := exp
dec -> vardec
decs ->
decs -> dec decs
decs -> dec decs
type-id -> id
exp -> num
l-value -> type-id [ exp ]
exp -> l-value
exp -> string
exp -> exp = exp
expseq1 ->
expseq -> exp expseq1
exp -> let decs in expseq end
```

#### RESULTADO:

ANALISE SINTATICA OK!

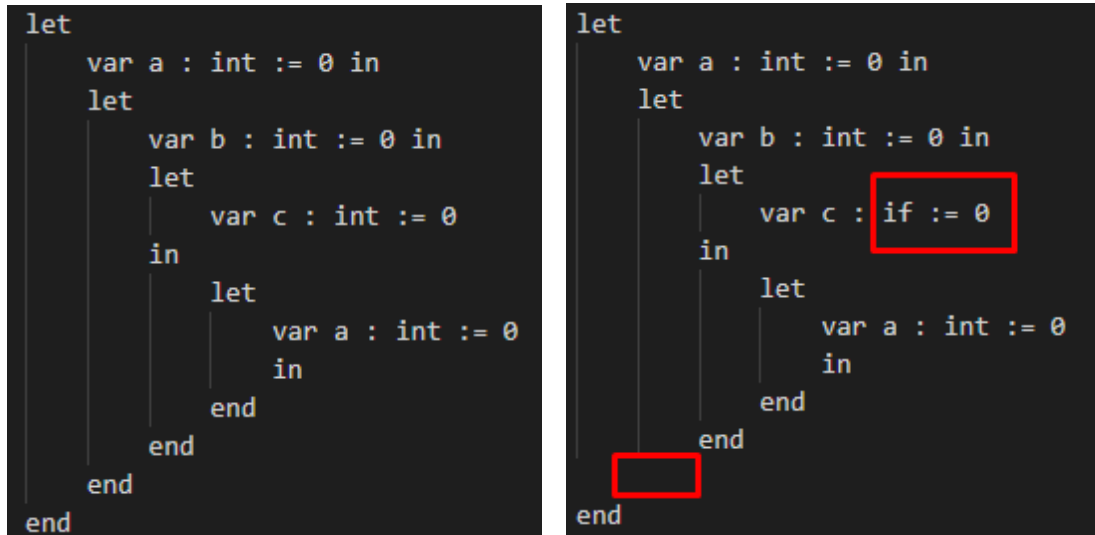
#### CÓDIGO FONTE SUBMETIDO:

```
let
  type tipoarranjo = array of int
  var Arranjo:tipoarranjo := tipoarranjo [10] of 0
in
  Arranjo[2] = "nome"
end
```

**Saída:** regras, resultado e listagem do código fonte

## 5.2. Exemplo com erro de sintaxe

Abaixo mostramos o código utilizado para teste de erro de sintaxe. A primeira imagem mostra o código como deveria ser, com a sintaxe correta. A segunda imagem mostra o mesmo código, porém com alguns erros de sintaxe inseridos propositalmente para que possamos testar a detecção de erros em nosso analisador.



```
let
  var a : int := 0 in
    let
      var b : int := 0 in
        let
          var c : int := 0
        in
          let
            var a : int := 0
          in
            end
        end
      end
    end
  end
end
```

```
let
  var a : int := 0 in
    let
      var b : int := 0 in
        let
          var c : if := 0
        in
          let
            var a : int := 0
          in
            end
        end
      end
    end
  end
```

Entrada: tests/06.txt

REGRAS UTILIZADAS:

```
type-id -> id
exp -> num
vardec -> var id : type-id := exp
dec -> vardec
decs ->
decs -> dec decs
type-id -> id
exp -> num
vardec -> var id : type-id := exp
dec -> vardec
decs ->
decs -> dec decs
```

RESULTADO:

ERRO DE SINTAXE!

CÓDIGO FONTE SUBMETIDO:

```
let
  var a : int := 0 in
    let
      var b : int := 0 in
        let
          var c : if := 0
        in
          let
            var a : int := 0
          in
            end
        end
      end
    end
  end
```

Saída: regras, resultado e listagem do código fonte

## 6. Instruções para utilização do analisador

1. Utilizar um sistema operacional Linux.
2. Instalar as ferramentas Flex, Bison e C:

```
sudo apt install flex  
sudo apt install bison
```

3. Ir para a pasta root do projeto.
4. Executar o seguinte comando para gerar o scanner:

```
make compile
```

5. Colocar o arquivo com o código fonte no diretório do analisador.
6. Executar o comando:

```
./a.out <diretorio_do_arquivo>/<nome_do_arquivo>
```

7. Caso o analisador não acuse erro de sintaxe, a saída será no formato:

```
"ANALISE SINTATICA OK!"
```

caso contrário, a saída será no formato:

```
"ERRO DE SINTAXE!"
```

8. Em ambos os casos, tanto as regras usadas pelo analisador, quanto o resultado da análise e o código fonte submetido, serão impressos no terminal.

## 7. Ferramentas de apoio

Como ferramentas de apoio para o desenvolvimento do analisador sintático, utilizamos as tecnologias Flex e Bison para gerar o código de análise léxica e dos arquivos resultantes da análise sintática.

## 8. Suposições sobre as especificações que não ficaram claras

- *“Apresentou as regras usadas para acompanhar o reconhecimento sintático dos programas de testes submetidos ao Analisador?”*

Entendemos que a apresentação dessas regras seria pela impressão das produções da gramática detectadas em cada parte do código fonte.

- “Apresentou os resultados dos programas de testes submetidos ao Analisador Sintático?”

Entendemos que a apresentação desses resultados seria pela impressão de uma simples frase dizendo se o código está com a sintaxe correta ou se está com erro de sintaxe.



- “Apresentou a listagem do código fonte submetido ao gerador de análise sintática?”

Entendemos que a apresentação do código fonte seria pela impressão do mesmo no terminal após o término da análise.

## 9. Conclusão

Com este trabalho, concluímos que a escrita de uma boa gramática para uma determinada linguagem é de extrema importância. Gramáticas ambíguas geram conflitos *shift-reduce* ou *reduce-reduce* e podem prejudicar completamente a fase da análise sintática. Também pudemos perceber qual é realmente a função do analisador sintático na construção de tabelas sintáticas LALR(1) para a linguagem.

## 10. Referências

1.  Part 01: Tutorial on lex/yacc
2.  Part 02: Tutorial on lex/yacc.
3. Aho, Alfred V., Lam Monic S., Sethi, R.; Ullman and Jeffrey D., Compilers Principles, Techniques, & Tools , 2nd Edition, Pearson Addison Wesley, New York, 2007
4. [https://www.math.utah.edu/docs/info/bison\\_8.html](https://www.math.utah.edu/docs/info/bison_8.html)