Bison

Instituto Superior de Engenharia do Porto

2021/2022

BISON

- O *BISON* é um gerador de analisadores sintáticos de âmbito genérico;
- Reconhece gramáticas independentes do contexto LALR(1) *Look-Ahead Left to right Rightmost derivation*;
- Gera um programa C/C++, capaz de processar frases da linguagem reconhecida pela gramática;

BISON

- O *BISON* é um gerador de analisadores sintáticos de âmbito genérico;
- Reconhece gramáticas independentes do contexto LALR(1) *Look-Ahead Left to right Rightmost derivation*;
- Gera um programa C/C++, capaz de processar frases da linguagem reconhecida pela gramática;
- É implementado através de um autómato de pilha que:
  - Insere os *tokens* no topo pilha do autómato, "*shift*";
  - Quando tem no topo da pilha os símbolos do lado direito de uma regra faz "*reduce*", isto é, transforma esses símbolos no não terminal (regra) que fica no topo da pilha.

BISON

- O *BISON* é um gerador de analisadores sintáticos de âmbito genérico;
- Reconhece gramáticas independentes do contexto LALR(1) *Look-Ahead Left to right Rightmost derivation*;
- Gera um programa C/C++, capaz de processar frases da linguagem reconhecida pela gramática;
- É implementado através de um autómato de pilha que:
  - Insere os *tokens* no topo pilha do autómato, "*shift*";
  - Quando tem no topo da pilha os símbolos do lado direito de uma regra faz "*reduce*", isto é, transforma esses símbolos no não terminal (regra) que fica no topo da pilha.
- O *BISON* também pode reconhecer gramáticas LR, gerando um *parser* GLR (Generalized LR);

Gramática *BISON*

Texto a analisar

```
int square(int x)
{ // square function
  return x * x;
}
```

Gramática *BISON*

Texto a analisar

```
int square(int x)
{ // square function
  return x * x;
}
```

Resultado da análise léxica

```
TIPO_INT ID '(' TIPO_INT ID ')'
'{'
  RETURN ID '*' ID ';'
'}'
```

Gramática *BISON*

Texto a analisar

```
int square(int x)
{ // square function
  return x * x;
}
```

Resultado da análise léxica

```
TIPO_INT ID '(' TIPO_INT ID ')'
'{'
  RETURN ID '*' ID ';'
'}'
```

```
funcao:  tipo ID '(' parametros ')'
         bloco_de_instrucoes
     ;
parametros: /* vazio */
       | lista_parametros
     ;
lista_parametros: parametro
       | lista_parametros ',' parametro
     ;
parametro: tipo ID
       ;
```

Gramática *BISON*

Texto a analisar

```
int square(int x)
{ // square function
  return x * x;
}
```

Resultado da análise léxica

```
TIPO_INT ID '(' TIPO_INT ID ')'
'{'
  RETURN ID '*' ID ';'
'}'
```

```
bloco_de_instrucoes: '{' instrucoes '}'
      ;
instrucoes: /* vazio */
        | instrucoes instrucao
      ;
instrucao: RETURN expressao ';'
       ;
tipo:       TIPO_INT | TIPO_FLOAT | TIPO_CHAR
       ;
```

Gramática *BISON*

Texto a analisar

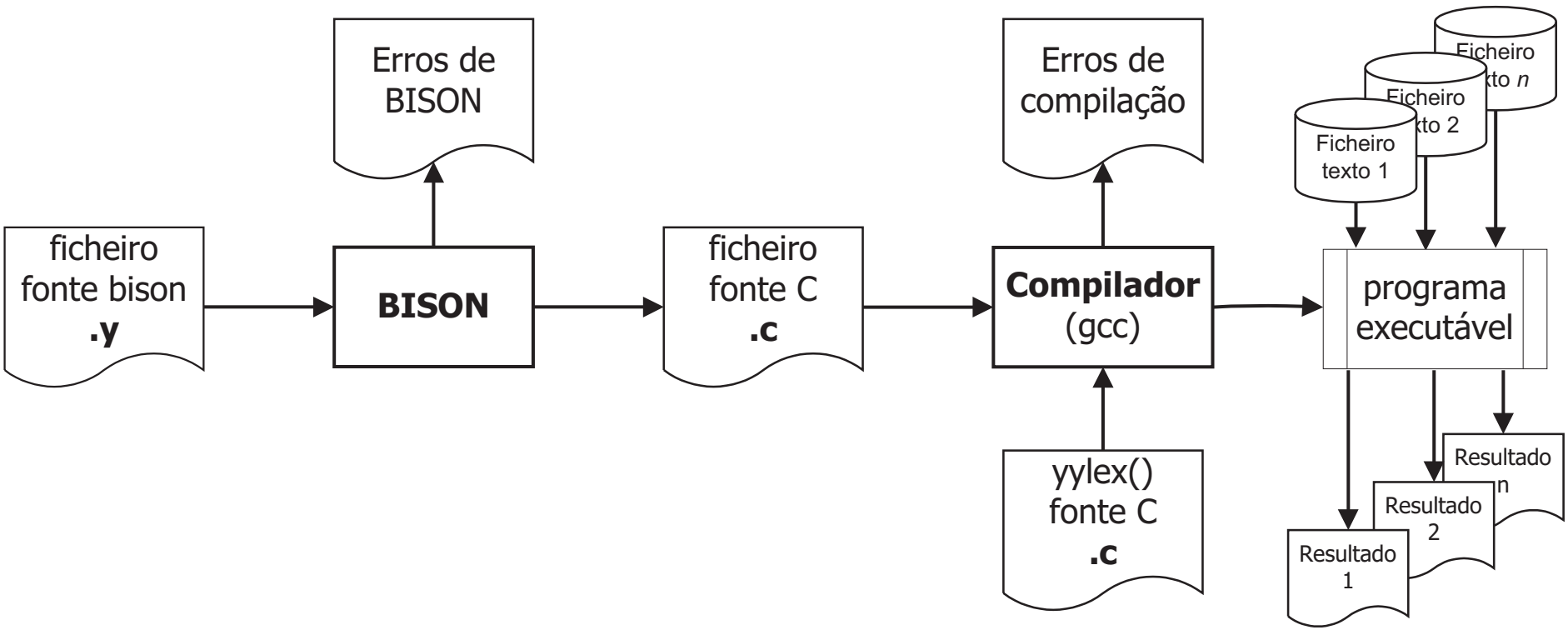
```
int square(int x)
{ // square function
  return x * x;
}
```

Resultado da análise léxica

```
TIPO_INT ID '(' TIPO_INT ID ')'
'{'
  RETURN ID '*' ID ';'
'}'
```

```
expressao:  operando resto_expressao
        ;
resto_expressao: /* vazio */
          | operador expressao
        ;
operando: INTEIRO | REAL | ID
          ;
operador: '+' | '-' | '*' | '/'
          ;
```

Funcionamento do *FLEX*.py:myBisonLexer -x



Exemplo básico dum ficheiro *BISON*

```
%{
  #include <stdio.h>
  int numArgs=0, numErros=0;
%}

%token ID INT REAL
%start inicio

%%
```

Exemplo básico dum ficheiro *BISON*

```
%%

inicio:       /* vazio */
            | lista_args
            ;
lista_args: arg
            | lista_args ',' arg
            ;
arg:        ID          {numArgs++;}
            | INT       {numArgs++;}
            | REAL      {numArgs++;}
            ;
%%
```

Exemplo básico dum ficheiro *BISON*

```
%%

int main(){
  yyparse();
  if(numErros==0)
    printf("Frase válida\n");
  else
    printf("Frase inválida\nNúmero de erros: %d\n",numErros);
    printf("Número de argumentos é %d\n",numArgs);
  return 0;
}
int yyerror(char *s){
  numErros++;
  printf("erro sintatico/semantico: %s\n",s);
}
```

Opções *BISON*

Valores semânticos

- Podemos usar a variável *yylval* para guardar os valores semânticos (lexemas) identificados pelo analisador léxico;
- Por omissão a variável *yylval* é definida como um inteiro;

Armazenamento do lexema no *FLEX*

```
[-+]?[0-9]+        {yylval=atoi(yytext);return INT;}
```

Opções *BISON*

Valores semânticos

- Podemos usar a variável *yylval* para guardar os valores semânticos (lexemas) identificados pelo analisador léxico;
- Por omissão a variável *yylval* é definida como um inteiro;

Armazenamento do lexema no *FLEX*

```
[-+]?[0-9]+        {yylval=atoi(yytext);return INT;}
```

- É possível redefinir o tipo da variável *yylval*:

Redefinição do tipo da variável *yylval*

```
#define YYSTYPE double
```

Opções *BISON*

Valores semânticos

- Quando há necessidade de mais de um tipo pode ser usada a definição de uma

  `%union`

;

Utilização da opção *%union*

```
%union {
    char  *id;
    int   inteiro;
    float real;
}
```

Opções *BISON*

Valores semânticos

- Quando há necessidade de mais de um tipo pode ser usada a definição de uma

  `%union`

;

Utilização da opção *%union*

```
%union {
    char  *id;
    int   inteiro;
    float real;
}
```

Atribuição do tipo aos *tokens* e regras

```
%token <id>      ID STRING
%token <inteiro> INT
%token <real>    REAL
%type  <real>    operando expressao
```

Opções *BISON*
Valores semânticos

Armazenamento dos lexemas no *FLEX*

```
%%
[-+]?[0-9]+ {
              yylval.inteiro=atoi(yytext);
              return INT;
            }
\"[^"\n]*\" {
              yylval.id=yytext;
              return STRING;
            }
[_a-zA-Z][_a-zA-Z0-9]* {
              yylval.id=strdup(yytext);
              return ID;
            }
%%
```

Opções *BISON*

Precedência de operadores

- A precedência de operadores no *BISON* serve para definir a ordem pela qual as regras alternativas são processadas, eliminando assim os conflitos que possam surgir.

Tipos de precedências do *BISON*

```
%left      OPL
%right     OPD
%nonassoc  OPNA
```

Exemplo

```
%left '<' '>' '=' DIF MEN_IG MAI_IG
%left '+' '-'
%left '*' '/'
%left '^'
%nonassoc MENOS_UNARIO
```

Opções *BISON*

Precedência de operadores

- A precedência de operadores no *BISON* serve para definir a ordem pela qual as regras alternativas são processadas, eliminando assim os conflitos que possam surgir.

Exemplo

```
expressao: expressao '+' expressao
      | expressao '-' expressao
      | expressao '*' expressao
      | expressao '/' expressao
      | '-' expressao %prec MENOS_UNARIO
      | operando
      ;
operando:  INTEIRO
      | REAL
      | ID
      ;
```

Opções *BISON*
Acções Semânticas

- O *BISON* pode ter acções semânticas ao longo das regras;
- Cada acção semântica ocupa um $ da regra dependendo da posição ocupada;
- O lado esquerdo (regra) é referenciado sempre por $$.

Acções semânticas intermédias *BISON*

```
expressao: {printf("antes da expressão ($1)");}
      INT {printf("antes do operador ($3)");}
      '+' {printf("depois do operador ($5)");}
      INT {printf("depois da expressão ($7)");}
      ;
```

Opções *BISON*

Acções Semânticas

- O *BISON* pode ter acções semânticas ao longo das regras;
- Cada acção semântica ocupa um $ da regra dependendo da posição ocupada;
- O lado esquerdo (regra) é referenciado sempre por $$.

Alternativa equivalente

```
expressao: ac1 ac2 ac3 ac4
          ;
          ac1:        /* vazio */ {printf("antes da expressao");}
          ;
ac2:       INT {printf("antes do operador");}
          ;
ac3:       '+' {printf("depois do operador");}
          ;
ac4:       INT {printf("depois da expressao");}
          ;
```

Opções *BISON*
Acções Semânticas

- O *BISON* pode ter acções semânticas ao longo das regras;
- Cada acção semântica ocupa um $ da regra dependendo da posição ocupada;
- O lado esquerdo (regra) é referenciado sempre por $$.

Exemplo de utilização dos valores semânticos

```
expressao: operando '+' operando {$$=$1+$3;}
         ;
operando:   INTEIRO   {$$=$1;}
         | REAL       {$$=$1;}
         ;
```

Gramática *BISON*

Recursividade à direita ou à esquerda?

- O *BISON* pode analisar gramáticas com recursividade à direita ou à esquerda;
- As gramáticas com recursividade à esquerda são mais eficientes.

Recursividade à direita *BISON*

```
lista: INT
     | INT ',' lista
     ;
```

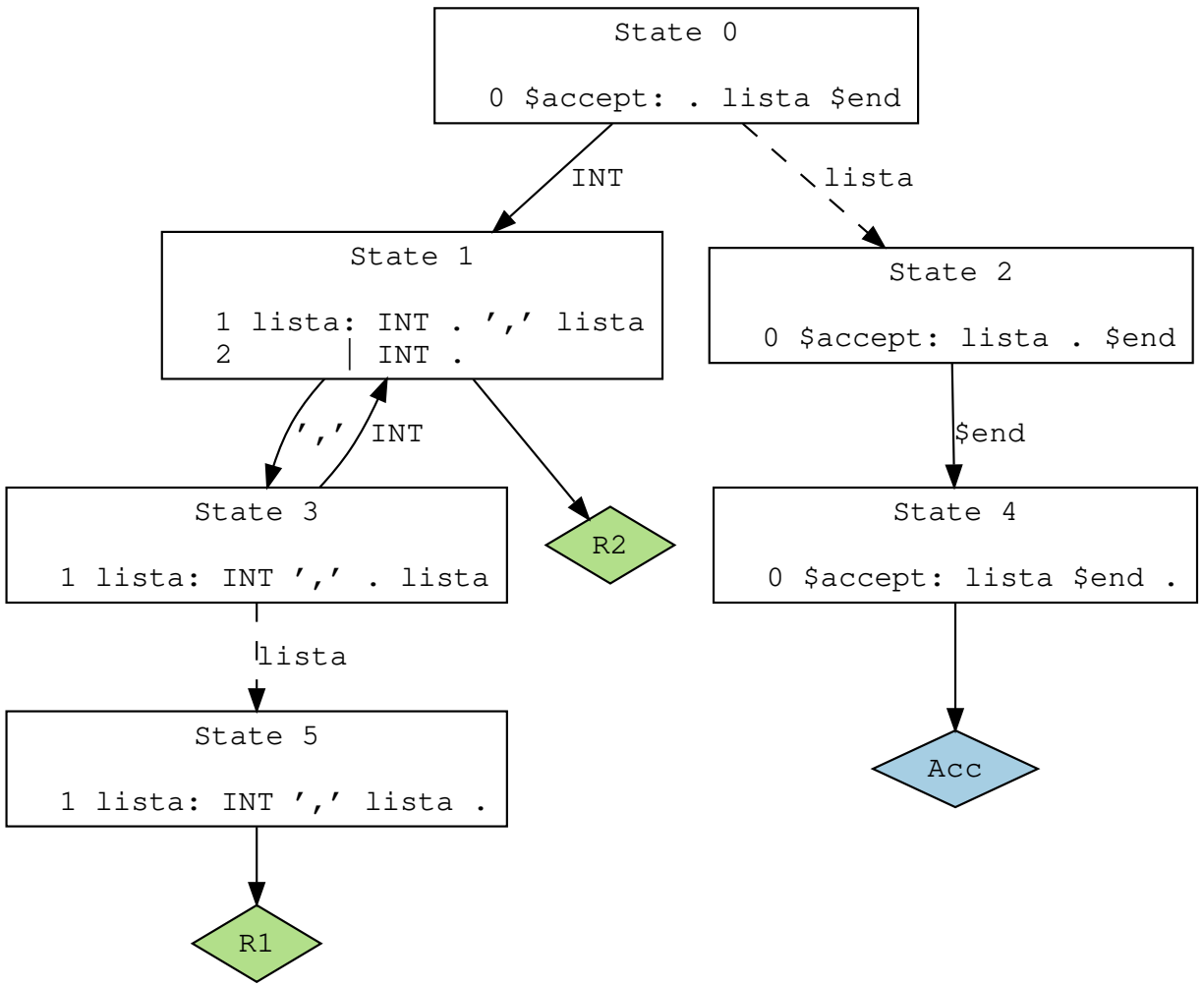
Recursividade à esquerda *BISON*

```
lista: INT
     | lista ',' INT
     ;
```

Gramática *BISON*

Recursividade à direita?

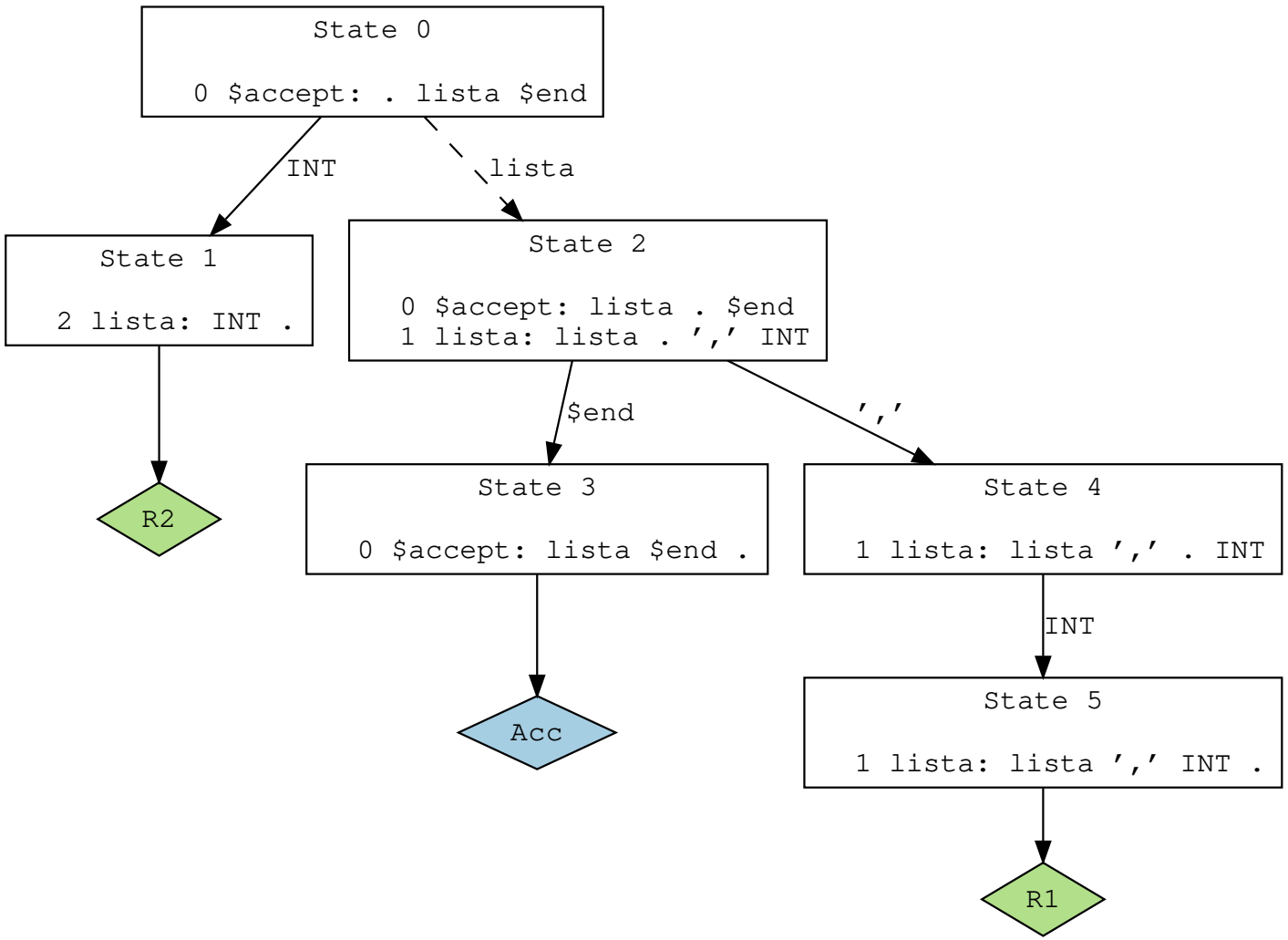
Fica a saltar entre os estados 1 e 3 até chegar ao último elemento da lista, só depois começa a fazer `reduces`. Preenche $n \times 2$ posições na *stack* e processa a lista pela ordem inversa.



Gramática *BISON*

Recursividade à esquerda?

Só ocupa 3 posições na *stack*, reduzindo depois para `lista`. Processa a lista pela ordem correcta.



Analisador léxico realizado com o *FLEX*

```
%{
  #include"exemplo.tab.h" /* header gerado pelo bison */
  extern int numErros;    /* vaiável criado no bison */
%}

%%
```

Analisador léxico realizado com o *FLEX*

```
%%

,                           return yytext[0];
[0-9]+                      return INT;
[0-9]+\.[0-9]+              return REAL;
[_a-zA-Z][_a-zA-Z0-9]*      return ID;
[ \t]                       /* ignorado */

.       {  printf("Erro lexico: simbolo desconhecido %s\n",yytext);
           numErros++;
        }
\n      /* ignorado, podia contar linhas */
<<EOF>>                     return 0;

%%
```

Conflitos *BISON*

reduce/reduce

- Quando o *BISON* pode realizar "*reduce*" a 2 regras simultâneamente gera um conflito *reduce/reduce*;
- Este erro surge normalmente da ambiguidade da gramática, e deve ser sempre corrigido.

```
sequencia: /* vazio */
    | talvez_palavra
    | sequencia palavra
        ;
talvez_palavra: /* vazio */
        | palavra
        ;
palavra: TOKEN
        ;
```

Conflitos *BISON*

shift/reduce

- Quando o *BISON* pode realizar "*shift*" de um token ou "*reduce*" a 1 regras simultâneamente gera um conflito *shift/reduce*;
- Este erro surge normalmente da ambiguidade da gramática;
- O *BISON* resolve este conflito usando sempre o "*shift*".

```
if_stmt:  IF expr THEN stmt
        | IF expr THEN stmt ELSE stmt
        ;
stmt:     TOKEN
        | if_stmt
        ;
expr:     TRUE
        | FALSE
        ;
```

Conflitos *BISON*

Output gerado com a opção -v

```
Regras nunca reduzidas
  4 talvez_palavra: /* vazio */

Estado 0 conflitos:
         1 de deslocamento/redução,
         2 de redução/redução

Gramática
    0 $accept: sequencia $end

    1 sequencia: /* vazio */
    2          | talvez_palavra
    3          | sequencia palavra

    4 talvez_palavra: /* vazio */
    5               | palavra

    6 palavra: TOKEN
```

Conflitos *BISON*

Output gerado com a opção -v

```
Terminais, com as regras onde eles aparecem
$end (0) 0
error (256)
TOKEN (258) 6

Não-terminais com as regras onde eles aparecem
$accept (4)
    à esquerda: 0
sequencia (5)
    à esquerda: 1 2 3, à direita: 0 3
talvez_palavra (6)
    à esquerda: 4 5, à direita: 2
palavra (7)
    à esquerda: 6, à direita: 3 5
```

Conflitos *BISON*

Output gerado com a opção -v

```
estado 0

 0 $accept: . sequencia $end

 TOKEN  deslocar, e ir ao estado 1

 $end    reduzir usando a regra 1 (sequencia)
 $end    [reduzir usando a regra 4 (talvez_palavra)]
 TOKEN   [reduzir usando a regra 1 (sequencia)]
 TOKEN   [reduzir usando a regra 4 (talvez_palavra)]
 $padrão reduzir usando a regra 1 (sequencia)

 sequencia       ir ao estado 2
 talvez_palavra  ir ao estado 3
 palavra         ir ao estado 4
```

Conflitos *BISON*

Output gerado com a opção -v

```
estado 10

    1 if_stmt: IF expr THEN stmt .
    2        | IF expr THEN stmt . ELSE stmt

    ELSE  deslocar, e ir ao estado 11

    ELSE     [reduzir usando a regra 1 (if_stmt)]
    $padrão  reduzir usando a regra 1 (if_stmt)
```

Recuperação de erros no *BISON*

- Permite ao *BISON* continuar a análise mesmo quando encontra um erro;
- Pode ser realizada com regras que "simulem" o erro;
- Pode ser realizada usando o "*token*" *error* que instância com qualquer sequência de "*tokens*".

```
lista_args: arg
      | lista_args ',' arg
      | lista_args     arg
            {yyerror("falta virgula");}
      | lista_args ',' error
            {yyerror("falta argumento");}
      | error
            {yyerror("erro grave, sem recuperação");}
      ;
```

Exemplo avançado dum ficheiro *BISON*

```
%{
    #include <stdio.h>
    int numArgs=0, numErros=0;
%}

%union {
  char  *id;
  int   inteiro;
  float real;
}
%token <id>      ID
%token <inteiro> INT
%token <real>    REAL
%start inicio

%%
```

Exemplo avançado dum ficheiro *BISON*

```
%%
inicio:       /* vazio */
            | lista_args
            ;
lista_args: arg
            | lista_args ',' arg
            | lista_args ',' error {yyerror("falta argumento");}
            | lista_args {yyerror("falta virgula");} arg
            ;
arg:        ID          {numArgs++;printf("ID:%s\n",$1);}
            | INT       {numArgs++;printf("INT:%d\n",$1);}
            | REAL      {numArgs++;printf("REAL:%f\n",$1);}
            ;
%%
```

Exemplo avançado dum ficheiro *BISON*

```
%%

int main(){

  yyparse();

  if(numErros==0)
    printf("Frase v lida\n");
  else
    printf("Frase inv lida\nN mero de erros: %d\n",numErros);
    printf("N mero de argumentos  %d\n",numArgs);
  return 0;
}

int yyerror(char *s){
  numErros++;
  printf("erro sintatico/semantico: %s\n",s);
}
```

Exemplo avançado dum ficheiro *FLEX*

```
%{
  #include"exemplo.tab.h" // header greado pelo bison
  extern int numErros;
%}


%%

,                       return yytext[0];
[0-9]+                  yylval.inteiro=atoi(yytext); return INT;
[0-9]+\.[0-9]+          yylval.real=atof(yytext); return REAL;
[_a-zA-Z][_a-zA-Z0-9]*  yylval.id=yytext; return ID;
```

Exemplo avançado dum ficheiro *FLEX*

```
[_a-zA-Z][_a-zA-Z0-9]*    yylval.id=yytext; return ID;
[ \t]                     /* ignorado */

.                         {
      printf("Erro lexico: simbolo desconhecido %s\n",yytext);
      numErros++;
    }

\n                        return 0;
<<EOF>>                   return 0;

%%
```