Compiladores
Roteiro de Laboratório 02 – Construindo *Parsers*

Parte I
Utilizando o `bison`

1 Introdução

- No conteúdo teórico do Módulo 02 vimos que o analisador sintático (*parser*) é o segundo componente do *front-end* de um compilador.
- *Parsers* podem ser gerados *automaticamente* através de uma ferramenta.
- O `bison` serve para gerar *parsers* em C.
- **Entrada**: arquivo de descrição do *parser*: `*.y`. Contém as regras da gramática e as ações a serem tomadas em cada derivação.
- **Saída**: programa na linguagem C que implementa o *parser* especificado. (Arquivo *default*: `parser.tab.c`).
- Um arquivo de especificação do `bison` possui três partes:

```
seção de definições
%%
regras da gramática (produções)
%%
funções auxiliares
```

- Ao lado do corpo de cada regra você pode colocar uma ação (trecho de código em C), que é executada sempre que a regra indicada é aplicada pelo *parser*.
- As regras de tradução têm a seguinte forma:

```
Cabeça : Corpo    { Ação }
```

- *Obs.:* Não é coincidência que o arquivo do `bison` tenha a mesma estrutura geral que o arquivo de entrada do `flex`. Afinal, ambas as ferramentas foram desenvolvidas para serem utilizadas em conjunto.

2 Utilizando o `bison`

Obs.: Todos os exemplos apresentados nesse roteiro estão disponíveis no arquivo `.zip` de exemplos deste laboratório.

2.1 Exemplo 01 – Uma gramática simples de somas de dígitos

- Considere o seguinte arquivo, `parser.y`.

```
%{
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
```

```
int yylex(void);
void yyerror(char const *s);
%}

%token DIGIT PLUS ENTER

%%

line: expr ENTER ;
expr: expr PLUS expr | DIGIT ;

%%

int yylex(void) {
    int c = getchar();
    if (isdigit(c))     { return DIGIT; }
    else if (c == '+')  { return PLUS; }
    else if (c == '\n') { return ENTER; }
    // EOF is not a token but a constant from stdio.
    else if (c == EOF)  { return EOF; }
    else { // Not a digit or plus or enter.
        printf("LEXICAL ERROR: Unknown symbol %c\n", c);
        exit(EXIT_FAILURE);
    }
}

int main(void) {
    if (yyparse() == 0) printf("PARSE SUCCESSFUL!\n");
    else                printf("PARSE FAILED!\n");
    return 0;
}
```

- O código do `bison` acima é uma simples gramática livre de contexto que reconhece somas de dígitos.
- O comando `%token` especifica os tipos de *token* reconhecidos pelo *parser*.
- As duas linhas na segunda seção do arquivo especificam as *três* regras da gramática: note que a linha que começa com `expr` possui *duas* regras (separadas por `|`)!
- A função `yylex()` é a função de conexão com o *scanner*.
  - Essa função retorna um inteiro que é a constante representando o *tipo* do *token*.
  - Nesse exemplo a função foi criada "na mão", mas na prática utilizamos o `flex` para criar o *scanner* correspondente (veja Exemplo 02).
- A função `yyerror()` é chamada quando é detectado algum erro de sintaxe. Nesse exemplo, vamos usar a implementação padrão dessa função que só imprime a mensagem `syntax error` quando o *parser* detecta uma entrada com sintaxe inválida. Em exemplos posteriores, vamos melhorar as mensagens de erro.
- Gerando o *parser* com o `bison`:

```
$ bison parser.y
parser.y: warning: 1 shift/reduce conflict [-Wconflicts-sr]
```

A mensagem de *shift/reduce* indica *ambiguidade* na gramática. Apesar de aparecer como *warning* isso na verdade é um **erro**! (Vamos consertar esse problema depois.)

- Compilando e executando o *parser*:

```
$ gcc -Wall parser.tab.c -ly
$ ./a.out <<< "2"
PARSE SUCCESSFUL!
$ ./a.out <<< "2+3"
PARSE SUCCESSFUL!
$ ./a.out <<< "2+3+5+7"
PARSE SUCCESSFUL!
$ ./a.out <<< "2+3+5+7+a"
LEXICAL ERROR: Unknown symbol a
$ ./a.out <<< "2 + 3"
LEXICAL ERROR: Unknown symbol
$ ./a.out <<< "42+1"
syntax error
PARSE FAILED!
```

- Não esqueça a opção `-ly` ao compilar, pois senão isso leva a um erro como abaixo.

```
$ gcc -Wall parser.tab.c
/usr/bin/ld: in function `yyparse':
parser.tab.c: undefined reference to `yyerror'
/usr/bin/ld: parser.tab.c: undefined reference to `yyerror'
collect2: error: ld returned 1 exit status
```

- A opção de compilação `-ly` faz a ligação com a biblioteca do `bison` que provê uma implementação padrão de `yyerror()`. Essa função é a responsável pela exibição da mensagem `syntax error` na execução acima.
- É importante notar que os dois últimos erros na execução do *parser* acima são causados pelas sérias limitações do *scanner*, que não reconhece espaços em branco e nem números com mais de um dígito. Para resolver esse problema, vamos usar o `flex` para implementar o *scanner*.

2.2 Exemplo 02 – Unindo `flex` e `bison`

- Modificando o exemplo anterior para usar o `flex` e aceitar números naturais com qualquer quantidade de dígitos.
- Arquivo `parser.y`:

```
%{
#include <stdio.h>
int yylex(void);
void yyerror(char const *s);
%}
%token NUMBER PLUS ENTER
%%
line: expr ENTER ;
expr: expr PLUS expr | NUMBER ;
%%
int main(void) {
```

```
    if (yyparse() == 0) printf("PARSE SUCCESSFUL!\n");
    else                printf("PARSE FAILED!\n");
    return 0;
}
```

- Arquivo `scanner.l`:

```
%option outfile="scanner.c"
%option noyywrap
%option nounput
%option noinput

%{
#include "parser.h" // For the token types from bison.
%}

%%

[0-9]+  { return NUMBER; }
"+"     { return PLUS; }
"\n"    { return ENTER; }
<<EOF>> { return EOF; }
" "     { /* ignore spaces */ }
.       { printf("LEXICAL ERROR: Unknown symbol %s\n", yytext);
          exit(EXIT_FAILURE); }
```

- Compilando e executando o *parser*:

```
$  bison -Wall --defines=parser.h -o parser.c parser.y
parser.y: warning: 1 shift/reduce conflict [-Wconflicts-sr]
$ flex scanner.l
$ gcc -Wall scanner.c parser.c -ly
$ ./a.out <<< "2 + 3"
PARSE SUCCESSFUL!
$ ./a.out <<< "267 + 3456 + 6"
PARSE SUCCESSFUL!
$ ./a.out <<< "267 + 3456 +   6 + a"
LEXICAL ERROR: Unknown symbol a
$ ./a.out <<< "42 +"
syntax error
PARSE FAILED!
```

- Os dois últimos testes acima ilustram erros léxicos e sintáticos, respectivamente. Note que o *parser* agora aceita expressões com espaços pois o *scanner* reconhece e descarta esses símbolos.
- Dado que são necessários vários comandos para gerar o *parser*, o uso de um `Makefile` é recomendado (veja Exemplo 03).
- Inspecione os arquivos `parser.h` e `parser.c` para ver o código gerado automaticamente pelo `bison`. O código desse arquivo implementa um autômato de pilha que reconhece a gramática livre de contexto especificada. Tente encontrar os trechos de código C do arquivo `.y` no arquivo `.c`. Vale destacar o conteúdo do arquivo `parser.h`, em particular:

```
enum yytokentype
  {
    NUMBER = 258,
    PLUS = 259,
    ENTER = 260
  };
```

que mostra a enumeração criada pelo `bison` definindo os tipos de *tokens* criados com o comando `%token`.

- Algumas opções úteis para se usar no arquivo `.y` do `bison`:

```
// File name of generated parser.
%output "parser.c"
// Produces a 'parser.h'
%defines "parser.h"
// Give proper error messages when a syntax error is found.
%define parse.error verbose
// Enable lookahead correction to improve syntax error handling.
%define parse.lac full
```

Vamos usar essas opções a partir do próximo exemplo.

2.3 Exemplo 03 – Um programa analisador de datas

- Neste exemplo vamos desenvolver um analisador de datas simples, que aceita datas no formato `dd/mm/aaaa`, isto é, dia/mês/ano, aonde dia e mês possuem dois dígitos e ano possui quatro dígitos.
- Conforme indicado nos *slides* da Aula 02, os *tokens* podem ter um *atributo* (também chamado de *valor léxico*) associado a eles. No `flex`, esse valor fica guardado na variável `yylval`.
- Veja o arquivo `scanner.l` deste exemplo:

```
%option outfile="scanner.c"
%option noyywrap
%option nounput
%option noinput
%{
#include "parser.h" // For the token types from bison.
%}
%%
[0-9]+  { yylval = atoi(yytext); return NUMBER; }
"/"     { return SLASH; }
"\n"    { return ENTER; }
" "     { /* ignore spaces */ }
.       { printf("LEXICAL ERROR: Unknown symbol %s\n", yytext);
          exit(EXIT_FAILURE); }
```

Note o código C acima que converte o lexema do *token* `NUMBER` em um número inteiro e armazena esse valor na variável `yylval`. Vamos usar essa variável no *parser* a seguir.

- Arquivo `parser.y` (fragmento):

```
%{
#include <stdio.h>
int yylex(void);
void yyerror(char const *s);
void test_date(int, int, int);
%}
%token NUMBER SLASH ENTER
%%
dates: %empty | dates date ENTER ;
date: NUMBER SLASH NUMBER SLASH NUMBER {test_date($1, $3, $5);};
%%
```

- O comando `%empty` acima representa a *string* vazia ϵ. Esse comando só está disponível na versão 3+ do Bison. Versões mais antigas usavam `/* empty */` para indicar uma regra com o corpo vazio.
- As duas regras com a cabeça `dates` indicam que uma entrada é válida se ela for composta por zero ou mais datas separadas por quebra de linha. Em outras palavras, poderíamos ter uma regra como abaixo.

```
dates: {date ENTER}* ;
```

  Infelizmente o `bison` não aceita esse tipo de notação (chamada de EBNF – *extended* BNF). Assim, para indicar repetições, é necessário utilizar recursão, como no código original acima. *Obs.:* O ANTLR (outro gerador de *parsers* para Java) aceita notação EBNF.
- Como dito acima, o *scanner* armazena o valor numérico dos *tokens* `NUMBER` que forem reconhecidos. Toda vez que um *token* de número é retornado pelo *scanner*, o *parser* copia o valor da variável léxica `yylval` para uma *variável semântica*. No `bison`, essas variáveis são indicadas pelo símbolo `$`.
- Todos os símbolos da gramática no `bison` possuem uma variável `$` associadas a eles. Para acessar o valor de cada símbolo do corpo de uma regra, começamos a contar os símbolos a partir de 1. Assim, a chamada da função `test_date` utiliza os valores inteiros associados aos três *tokens* `NUMBER` da regra (variáveis `$1`, `$3` e `$5`). As variáveis `$2` e `$4` estão associadas aos *tokens* `SLASH`. Como o *scanner* não armazena nenhum valor para esse tipo de *token*, essas variáveis contém lixo de memória. Assim, devemos ter muito cuidado ao acessar as variáveis `$` no `bison`!
- No momento, só vamos mostrar o uso das variáveis semânticas associadas aos *tokens*. No futuro, vamos usá-las também com os símbolos não-terminais da gramática (próximos laboratórios).
- Compilando e executando o *parser*:

```
$ make
bison parser.y
flex scanner.l
gcc -Wall scanner.c parser.c -ly
Done.
$ ./a.out < tests
Testing date: 2/8/1930
Valid date!
Testing date: 0/1/9999
Invalid day: 0!
```

```
Testing date: 31/2/2016
Valid date!
syntax error, unexpected SLASH, expecting NUMBER
PARSE FAILED!
```

Note como o uso do `Makefile` simplifica a compilação. A última entrada que produz um erro sintático é `22//4/1970`, daí a mensagem de erro do *parser*. Foram utilizadas as opções adicionais do `bison` indicadas anteriormente, por isso a mensagem de erro mais detalhada. Convém destacar que o *parser* reconhece o dia 31 de fevereiro simplesmente porque a implementação da função `test_date` é muito simplória (veja o código no arquivo de exemplo). Não vamos tentar resolver esse problema porque ele não tem nada a ver com a construção de um *parser*, é somente uma questão de lógica de programação.

2.4 Exercícios de aquecimento

AVISO: Alguns dos exercícios abaixo possuem soluções nas próximas seções. Você é fortemente encorajado a parar agora para tentar resolver essas questões por conta própria, antes de continuar a leitura desse roteiro ou olhar as soluções do professor.

0. Faça o download dos arquivos de exemplo. Compile-os e execute-os como explicado acima. Usando o `bison` e o `flex`, crie e teste os *parsers* pedidos: (*Obs.:* Trabalhe somente com `stdin` e `stdout`. Não é preciso ficar abrindo e fechando arquivos. Use redirecionamentos do *shell* como exemplificado acima.)
1. Implemente um *parser* para o reconhecimento de parênteses pareados segundo a gramática abaixo (apresentada nos *slides* da Aula 02).

$$E \rightarrow (E) \mid a$$

2. Modifique o Exemplo 02 para reconhecer as quatro operações aritméticas básicas. As expressões podem ter parênteses. Veja a gramática abaixo (apresentada nos *slides* da Aula 02). *Obs.:* Não é necessário consertar os erros de *shift/reduce* porque nós ainda não aprendemos como fazer isso.

$$\begin{aligned} E &\rightarrow E\ O\ E \mid (E) \mid \text{num} \\ O &\rightarrow + \mid - \mid * \mid / \end{aligned}$$

3. Implemente um *parser* para o reconhecimento de comandos *if-then-else* segundo a gramática abaixo (apresentada nos *slides* da Aula 02).

statement → ***if-stmt*** | **`other`**
if-stmt → **`if (`** ***exp*** **`)`** ***statement***
| **`if (`** ***exp*** **`)`** ***statement*** **`else`** ***statement***
exp → **`0`** | **`1`**

Obs.: Novamente não é necessário consertar os erros de *shift/reduce*. O seu *parser* deve aceitar entradas como abaixo.

```
other
if (0) other
if (1) other
if (0) other else other
if (1) other else other
if (0) if (1) other else other
```

E rejeitar entradas como:

```
if (0) if (1) other if
```

4. Modifique o Exemplo 02 para realizar as operações de soma indicadas pela expressão de entrada. Ao final, imprima o resultado da soma caso a entrada seja sintaticamente válida. Por exemplo:

```
$ ./ex04 <<< "2 + 3 + 42"
PARSE SUCCESSFUL! Result = 47
```

Dica: Use uma variável global como acumulador dos valores dos números que chegam do *scanner*.

3 Removendo ambiguidades de gramáticas no `bison`

- Nas aulas teóricas e nos exemplos anteriores vimos que muitas gramáticas de interesse são *ambíguas*.
- Gramáticas com problemas de ambiguidade geram conflitos de *shift/reduce* no `bison`.
- Agora vamos aprender a usar algumas opções do `bison` para remoção de ambiguidades em gramáticas.

3.1 Exemplo 04 – Consertando o Exemplo 01

- A gramática do Exemplo 01 apresentada na seção anterior causa 1 conflito de *shift/reduce* no `bison`. (Os termos *shift* e *reduce* indicam que o `bison` é um *parser bottom up*.) Como dito acima, o problema é causado por uma ambiguidade na gramática.
- Para entender o que está acontecendo, devemos pensar nas possíveis *parse trees* que o `bison` pode construir para uma entrada.
- Suponha a entrada `2+3+4`. Quantas *parse trees* existem?
- Tudo depende da *associatividade* da operação de soma. Se a operação é associativa à esquerda, temos uma árvore que é equivalente à expressão `(2+3)+4`. Por outro lado, se a associatividade for à direita, a árvore construída será equivalente a `2+(3+4)`.
- É claro que nesse caso não faz diferença qual das duas árvores é utilizada, pois a operação de adição também é *comutativa*, mas o `bison` não sabe disso.
- Em praticamente todas as linguagens de programação, assume-se que o operador de soma é associativo à esquerda. Para informar isso ao `bison`, usamos o comando `%left`. (Existe o comando dual `%right` para os operadores com associatividade à direita.)
- Então, a única modificação necessária no Exemplo 01 é a inclusão de uma linha no arquivo `parser.y`, como abaixo:

```
%left PLUS  // Soma eh associativa a esquerda.
```

- Compilando o Exemplo 04 vemos que o conflito de *shift/reduce* foi removido.
- Teste esse novo *parser* com as mesmas entradas do Exemplo 02 para se certificar que ele continua funcionando como esperado.

3.2 Exemplo 05 – Consertando o Exercício 02

- Utilizando as informações do Exemplo 04, podemos ficar tentados a incluir uma linha como

```
%left PLUS MINUS TIMES OVER
```

no *parser* para eliminar os conflitos de *shift/reduce* da gramática do Exercício 02. Embora um comando como acima de fato remova esses conflitos, ele ainda não é adequado.

- Utilizando um comando `%left` como acima, estamos dizendo para o `bison` que todos os operadores são associativos à esquerda (correto) e que todos têm a mesma prioridade (errado!).
- Vamos supor uma entrada como `2+3*4`. Segundo a explicação do item anterior, a expressão será interpretada como `(2+3)*4`, quando o correto é `2+(3*4)`.
- Para consertar esse problema, devemos declarar o comando `%left` em diferentes linhas, aonde cada linha determina grupos de operadores com um mesmo nível de prioridade. Os grupos devem ser declarados em ordem *crescente* de prioridade.
- Veja o arquivo `parser.y` deste exemplo com as correções abaixo:

```
%token ENTER NUMBER PLUS MINUS TIMES OVER
%left PLUS MINUS    // Ops associativos a esquerda.
%left TIMES OVER    // Mais para baixo, maior precedencia.

%%

line: expr ENTER ;
expr:
  expr PLUS expr
| expr MINUS expr
| expr TIMES expr
| expr OVER expr
| NUMBER ;
```

- Compile esse exemplo para ver que não há mais conflitos de *shift/reduce* e aproveite para testar algumas expressões simples que envolvam todos os quatro operadores.

3.3 Exemplo 06 – Consertando o Exercício 03

- A solução do Exercício 03 possui um conflito de *shift/reduce* porque a gramática utilizada ilustra o problema do `else` *pendente*, que causa ambiguidade.
- Considere abaixo as regras que definem a sintaxe de um comando `if`.

```
ifstmt:
  IF LPAR expr RPAR stmt
| IF LPAR expr RPAR stmt ELSE stmt ;
```

Para se eliminar a ambiguidade dessas regras, convencionou-se que o `else` sempre deve ser associado ao `if` mais próximo. Isso corresponde a sempre dar prioridade para a segunda regra, quando possível.

- Assim, devemos criar níveis de prioridade novamente, mas nesse caso não faz sentido usarmos os comandos `%left` ou `%right`, pois não estamos tratando de associatividade, somente de precedência. Nesses casos, utiliza-se o comando `%precedence`.
- A precedência de uma regra no `bison` por *default* tem a precedência do seu último *token*. Assim, a precedência da primeira regra é equivalente à precedência de `RPAR`.
- Para que a segunda regra sempre tenha prioridade sobre a primeira, precisamos que a precedência do *token* `ELSE` seja maior que a do *token* `RPAR`.

- Seguindo esse raciocínio, a declaração de *tokens* deve ficar como abaixo:

```
%token ENTER LPAR RPAR ZERO ONE IF ELSE OTHER
%precedence RPAR
%precedence ELSE
```

- Compilando e executando o *parser*, obtemos os resultados esperados.

```
$ bison parser.y
$ ./parser < tests_OK
Parse successful!
$ ./parser < tests_BAD
syntax error, unexpected IF, expecting ENTER or ELSE
Parse failed...
```

3.4 Exemplo 07 – Operadores unários

- Para terminar, precisamos aprender como lidar com operadores *unários*, isto é, operadores que possuem somente um operando.
- Até agora, os quatro operadores aritméticos básicos utilizados são todos *binários*, mas precisamos pelo menos de um operador unário para podermos definir números negativos.
- Para ilustrar essa necessidade, considere um comando de atribuição como `x = -1`. Até agora, não temos condição de escrever uma expressão como essa, pois só temos o operador binário de subtração. Assim, seria obrigatório escrever algo como `x = 0 - 1` para se colocar um valor negativo em `x`. Isso obviamente não é uma forma adequada.
- O grande problema nesse caso é que temos duas operações: subtração (binária) e negação (unária) sendo representadas pelo mesmo símbolo `-` (*token* `MINUS`), e com prioridades diferentes. Para resolver esse caso, vamos definir um novo *token* `UMINUS` que será usado somente para criar um novo nível de prioridade do menos unário.
- Agora, vamos novamente modificar o Exemplo 05 para incluir operações de exponenciação (`^`) e menos unário. Exponenciação tem a maior prioridade e é associativa à direita. Menos unário tem a segunda maior prioridade e não possui associatividade. A definição dos *tokens* fica como abaixo.

```
%token ENTER NUMBER PLUS MINUS TIMES OVER POW
%left PLUS MINUS    // Ops associativos a esquerda.
%left TIMES OVER    // Mais para baixo, maior precedencia.
%precedence UMINUS // Menos unario mais precedencia que binario.
%right POW          // Exponenciacao eh associativa a direita.
```

- Vale destacar que o menos unário não foi declarado como um *token* na primeira linha acima porque ele não é de fato um *token* retornado pelo *scanner*. Ele foi criado somente para introduzir um novo nível de prioridade entre o operador `^` e os operadores `*` e `/`.
- Resta agora definir as regras, que ficam assim:

```
expr:
  expr PLUS expr
| expr MINUS expr
| expr TIMES expr
| expr OVER expr
| MINUS expr %prec UMINUS
```

```
| expr POW expr
| NUMBER ;
```

- O ponto fundamental no código acima é o novo comando `%prec` que foi utilizado na regra do menos unário. Esse comando simplesmente diz para o `bison` que a regra possui a mesma precedência do "*token*" `UMINUS`. Procure entender porque os *tokens* `MINUS` e `UMINUS` são utilizados nessa ordem na regra da gramática.
- Compile esse exemplo e teste algumas expressões simples que envolvam exponenciação e menos unário.

Parte II
Construindo um *parser* para EZLang

A tarefa deste laboratório é construir um *parser* para a linguagem EZLang, e portanto, a sua estrutura sintática deve ser devidamente descrita. Em outras palavras, é necessário definir quais são as regras da gramática livre de contexto que define a sintaxe de programas válidos.

4 Convenções sintáticas da linguagem EZLang

A gramática da linguagem está apresentada em notação BNF abaixo, aonde os terminais (*tokens*) estão escritos em CAIXA ALTA e os não-terminais em caixa baixa.

```
program -> PROGRAM ID SEMI vars-sect stmt-sect
vars-sect -> VAR opt-var-decl
opt-var-decl -> ϵ | var-decl-list
var-decl-list -> var-decl-list var-decl | var-decl
var-decl -> type-spec ID SEMI
type-spec -> BOOL | INT | REAL | STRING
stmt-sect -> BEGIN stmt-list END
stmt-list -> stmt-list stmt | stmt
stmt -> if-stmt | repeat-stmt | assign-stmt | read-stmt | write-stmt
if-stmt -> IF expr THEN stmt-list END
         | IF expr THEN stmt-list ELSE stmt-list END
repeat-stmt -> REPEAT stmt-list UNTIL expr
assign-stmt -> ID ASSIGN expr SEMI
read-stmt -> READ ID SEMI
write-stmt -> WRITE expr SEMI
expr -> expr LT expr
      | expr EQ expr
      | expr PLUS expr
      | expr MINUS expr
      | expr TIMES expr
      | expr OVER expr
      | LPAR expr RPAR
      | TRUE
      | FALSE
      | INT_VAL
      | REAL_VAL
```

```
      | STR_VAL
      | ID
```

A gramática acima possui ambiguidades que levam a conflitos de *shift-reduce*. Utilize os comandos do `bison` apresentados na seção anterior para remover as ambiguidades. Considere a seguinte ordem crescente de prioridade dos operadores binários: operadores de comparação (= e <) têm a menor prioridade, seguidos de + e −, e finalmente ∗ e / com a maior prioridade.

A partir da gramática acima, podemos ver rapidamente algumas características e simplificações de EZLang. Por exemplo, a regra de `var-decl` nos mostra que só é possível declarar uma variável de cada vez, pois o corpo da regra (`type-spec ID SEMI`) não indica uma repetição do *token* `ID`. Isso quer dizer que é sintaticamente incorreto declarações *à la* C como `int x,y;`. Certifique-se que você entendeu todas as regras da gramática antes de continuar.

5 Implementado um *parser* para a linguagem EZLang

As convenções léxicas da linguagem EZLang já foram apresentadas no roteiro do Laboratório 01. Utilize o mesmo *scanner* desenvolvido na tarefa passada para compor o seu *parser* + *scanner*. *Obs.:* serão necessárias algumas adaptações no arquivo `.l` do *scanner*, conforme indicado nas seções anteriores.

5.1 Entrada e saída do *parser*

O programa a ser analisado é lido da entrada padrão (`stdin`). Se o programa estiver correto, o seu *parser* deve exibir uma mensagem indicando que o programa foi aceito.

```
$ ./lab02 < program.ezl
PARSE SUCCESSFUL!
```

Se o programa possuir erros léxicos, exiba a mesma mensagem de erro do laboratório anterior. Por outro lado, se o programa possuir erros sintáticos, exiba uma mensagem informativa como:

```
$ ./lab02 < program.ezl
SYNTAX ERROR (XX): syntax error, unexpected UT, expecting ET
```

Aonde `UT` e `ET` são os tipos de *tokens* lido e esperado, respectivamente. Utilizando as opções `%define parse.error verbose` e `%define parse.lac full`, a mensagem depois do sinal de `:` já é gerada automaticamente pelo `bison`. Neste caso, basta definir a função de erro:

```
// Primitive error handling.
void yyerror (char const *s) {
    printf("SYNTAX ERROR (%d): %s\n", yylineno, s);
    exit(EXIT_FAILURE);
}
```

Algumas observações importantes:

- Note que quando definimos a função `yyerror` não é mais necessário executar o `gcc` com a opção `-ly`.
- O seu *parser* pode terminar a execução ao encontrar o primeiro erro no programa de entrada.
- Os programas de entrada para teste são os mesmos do laboratório anterior (`in.zip`). As saídas esperadas desta tarefa estão no arquivo `out02.zip`, disponível no AVA.