Laboratório 04 - Funcionalidades do `bison`

Prof. Eduardo Zambon

Departamento de Informática (DI)
Centro Tecnológico (CT)
Universidade Federal do Espírito Santo (UFES)

2016/2 – Compiladores
***Compiler Construction* (CC)**

Introdução

- No laboratório anterior vimos como usar o `bison` para construir *parsers* básicos que reconhecem estruturas sintáticas simples.
- Na aula teórica vimos que muitas gramáticas de interesse são ambíguas.
- **Aula de hoje**: opções do `bison` para remoção de ambiguidades em gramáticas.
- Também vamos estudar as ações semânticas que podem ser associadas às regras.

Exemplo 01 – Soma de Inteiros

```
%{ ...
%}

%token NUMBER PLUS ENTER

%%
line: expr ENTER ;
expr: expr PLUS expr | NUMBER ;

%%
int main(void) {
  int result = yyparse();
  if (result == 0) printf("Parse successful!\n");
  else printf("Parse failed...\n");
}
```

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

Qual é o problema? Gramática é ambígua: existem duas possíveis *parse trees* para cada expressão.

Exemplo 02 – Consertando o Exemplo 01

```
%{ ...
%}

%token NUMBER ENTER
%left PLUS

%%
line: expr ENTER ;
expr: expr PLUS expr | NUMBER ;

%%
int main(void) {
  int result = yyparse();
  if (result == 0) printf("Parse successful!\n");
  else printf("Parse failed...\n");
}
```

```
$ bison parser.y
```

Comando `%left` diz que `PLUS` é associativo à esquerda.

Exemplo 03 – Gramática de Expressões Aritméticas

```
%token NUMBER
%%
lines:
  %empty
| lines expr '\n'        { printf(">> %d\n", $2); } ;
expr:
  expr '+' expr          { $$ = $1 + $3; }
| expr '-' expr          { $$ = $1 - $3; }
| expr '*' expr          { $$ = $1 * $3; }
| expr '/' expr          { $$ = $1 / $3; }
| NUMBER;
%%
```

```
$ bison parser.y
parser.y: warning: 16 shift/reduce conflicts [-Wconflicts-sr]
```

```
$ gcc -Wall parser.c -ly
$ ./a.out
4+2
>> 6
4*2-2
>> 0
```

Regras Semânticas

- Cada linha é uma produção. Entre `{ }`: ação semântica.
- `$$ $1 $2`: valores semânticos dos elementos da regra.
- As variáveis `$` associam um valor a cada elemento da regra. `$$` é o valor da cabeça.
- Para os *tokens*, o valor vem de `yylval`, criado pelo *scanner*.
- Tipo padrão é inteiro.

Exemplo 04 – Consertando o Exemplo 03

```
%token NUMBER
%%
lines: %empty
| lines expr '\n'          { printf(">> %d\n", $2); } ;
expr: factor               /* Default: $$ = $1; */
| expr '+' factor          { $$ = $1 + $3; }
| expr '-' factor          { $$ = $1 - $3; } ;
factor: term
| factor '*' term          { $$ = $1 * $3; }
| factor '/' term          { $$ = $1 / $3; } ;
term: NUMBER ;
%%
```

```
$ bison parser.y
$ gcc -Wall parser.c -ly
$ ./a.out
4+2
>> 6
4*2-2
>> 6
```

Exemplo 05 – Consertando o Exemplo 03 [x2]

```
%token NUMBER
%left '+' '-'  /* Ops associativos a esquerda. */
%left '*' '/'  /* Mais para baixo, maior precedencia. */
%%
lines: %empty
| lines expr '\n'        { printf(">> %d\n", $2); } ;
expr: NUMBER
| expr '+' expr          { $$ = $1 + $3; }
| expr '-' expr          { $$ = $1 - $3; }
| expr '*' expr          { $$ = $1 * $3; }
| expr '/' expr          { $$ = $1 / $3; } ;
%%
```

```
$ bison parser.y
$ gcc -Wall parser.c -ly
$ ./a.out
4+2
>> 6
4*2-2
>> 6
```

Exemplo 06 – Tipo de `yylval`

Modificando o exemplo anterior para aceitar números de ponto flutuante. Arquivo `parser.y`:

```
%define api.value.type {double} // Tipo da variavel yylval

%token NUMBER

%left '+' '-'  /* Ops associativos a esquerda. */
%left '*' '/'  /* Mais para baixo, maior precedencia. */

%%
lines: %empty
| lines expr '\n'        { printf(">> %.2f\n", $2); } ;
expr: NUMBER
| expr '+' expr          { $$ = $1 + $3; }
| expr '-' expr          { $$ = $1 - $3; }
| expr '*' expr          { $$ = $1 * $3; }
| expr '/' expr          { $$ = $1 / $3; } ;
%%
```

Exemplo 06 – Tipo de `yylval`

Arquivo `scanner.l`:

```
%option outfile="scanner.c"
%option noyywrap
%option nounput
%option noinput
%{
#include "parser.h"
%}
number [0-9]+\.?|[0-9]*\.[0-9]+
%%
[ ]       ; // skip blanks
{number} { yylval = atof(yytext); return NUMBER; }
\n|.      { return yytext[0]; }
```

```
$ bison parser.y
$ flex scanner.l
$ gcc -Wall -o parser scanner.c parser.c -ly
$ ./parser
4.2 + 3.1
>> 7.3
```

Exemplo 07 – Problema do *Dangling Else*

A gramática apresentada no slide 42 da Aula 03 listada abaixo é ambígua.

```
%token ENTER LPAREN RPAREN ZERO ONE IF ELSE OTHER
%%
stmts: %empty | stmts stmt ENTER;
stmt: ifstmt | OTHER;
ifstmt:
  IF LPAREN expr RPAREN stmt
| IF LPAREN expr RPAREN stmt ELSE stmt;
expr: ZERO | ONE;
```

```
$ bison parser.y
parser.y: warning: 1 shift/reduce conflict [-Wconflicts-sr]
$ ./parser < tests_OK
Parse successful!
$ ./parser < tests_BAD
syntax error, unexpected IF, expecting ENTER or ELSE
Parse failed...
```

Exemplo 08 – Consertando o Exemplo 07

É possível fazer o `else` ser sempre associado ao `if` mais próximo dando uma precedência maior ao *token* `else`.

```
%token ENTER LPAREN  ZERO ONE IF OTHER
%precedence RPAREN
%precedence ELSE
%%
stmts: %empty | stmts stmt ENTER;
stmt: ifstmt | OTHER;
ifstmt:
  IF LPAREN expr RPAREN stmt
| IF LPAREN expr RPAREN stmt ELSE stmt;
expr: ZERO | ONE;
```

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

Exercício 0

a. Faça o download dos arquivos de exemplo. Compile-os e execute-os como explicado aqui.

b. Usando o `bison` + `flex`, crie e teste os *parsers* pedidos a seguir.

Exercício 1

Crie uma calculadora com as quatro operações básicas que aceita números inteiros em formato decimal e hexadecimal. Use a função `strtol` para converter um *token* no valor para `yylval`.

Exercício 2

Modifique a calculadora do Exercício 1 para incluir parênteses e as 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.

Exercício 3

Modifique a calculadora do Exercício 2 para incluir uma expressão condicional do tipo

```
if (exp == exp) exp else exp
```

com a semântica e precedência usuais.

Avalie a entrada

```
if (0 == 0) 2 else 9 + 1
```

Certifique-se que `==` é tratado como um único *token*. O seu *parser* deve rejeitar a entrada abaixo.

```
if (1 = = 2) 5 else 4
```

Exercício 4

Modifique a calculadora do Exercício 3 para permitir variáveis de uma única letra. Variáveis são inicializadas por ‘atribuições’ da forma `x = 42`. Variáveis podem aparecer em qualquer expressão, inclusive testes condicionais.

Laboratório 04 - Funcionalidades do `bison`

Prof. Eduardo Zambon

Departamento de Informática (DI)
Centro Tecnológico (CT)
Universidade Federal do Espírito Santo (UFES)

2016/2 – Compiladores
***Compiler Construction* (CC)**