Laboratório 03 - Construindo *Parsers*

Prof. Eduardo Zambon

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

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

Introdução

- Nas aulas em sala vimos que o *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*: `xxx.tab.c`).

Arquivo de entrada do `bison`

Um arquivo de especificação do `bison` possui três partes:

```
%{
prólogo
%}
seção de declarações
%%
regras da gramática (produções)
%%
epílogo
```

As regras de tradução têm a seguinte forma:

```
Padrão   { Ação }
```

Exemplo 01 – Gramática de Somas de Dígitos

```
%{
#include <ctype.h>
#include <stdio.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; }
  return c; // Not a digit or plus or enter.
}
int main(void) {
  int result = yyparse();
  if (result == 0) printf("Parse successful!\n");
  else printf("Parse failed...\n");
}
```

Exemplo 01 – Gramática de Somas de Dígitos

- `%token` especifica o tipo do *token*.
- Cada linha na segunda seção do arquivo é uma produção.
- `yylex()`: função de conexão com o *scanner*.
  - Retorna um inteiro que é a constante representando o tipo do *token*.
  - Criada “na mão” nesse exemplo. Na prática: `flex`.
- `yyerror()` função que é chamada quando é detectado erro de sintaxe.

Exemplo 01 – Gramática de Somas de Dígitos

Compilando o *parser*:

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

Mensagem de *shift/reduce* indica ambiguidade na gramática.
Apesar de aparecer como *warning* isso é um erro!

```
$ 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"
syntax error
Parse failed...
$ ./a.out <<< "2 + 3"
syntax error
Parse failed...
```

`-ly`: provê uma implementação padrão de `yyerror()`.

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

Exemplo 02 – Unindo `flex` e `bison`

Arquivo `scanner.l`:

```
%option outfile="scanner.c"
%option noyywrap
%option nounput
%option noinput
%{
#include "parser.h"
%}
%%
[0-9]+   { return NUMBER; }
"+"      { return PLUS; }
"\n"     { return ENTER; }
" "      ; // skip blanks
.        { return yytext[0]; }
```

Exemplo 02 – Unindo `flex` e `bison`

```
$ bison -Wall --defines=parser.h -o parser.c parser.y
parser.y: warning: 1 shift/reduce conflict [-Wconflicts-sr]
$ flex scanner.l
$ gcc -Wall -o parser scanner.c parser.c -ly
$ ./parser <<< "2 + 3"
Parse successful!
$ ./parser <<< "267 + 3456 +   6"
Parse successful!
$ ./parser <<< "267 + 3456 +   6 + a"
syntax error
Parse failed...
```

Uso de `Makefile` é recomendado.

Opções Úteis 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
```

Exemplo 03 – Analisador de Datas

Arquivo `scanner.l`:

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

Armazena o valor do *token* `NUMBER` na variável `yylval`;

Exemplo 03 – Analisador de Datas

```
%{
#include <stdio.h>
int yylex(void);
void yyerror(char const *s);
void test_date(int, int, int);
%}
%token NUMBER
// Tell bison the lexemes for these token types so we can use
// them in the grammar rules.
%token SLASH "/"
%token ENTER "\n"
%%
dates:
  %empty
| dates date "\n" ;
date:
  NUMBER "/" NUMBER "/" NUMBER { test_date($1, $3, $5); };
%%
```

`%empty` representa a string vazia ϵ.

- Disponível somente na versão do Bison 3+.
- Para versão antiga use: `/* empty */`.

Exercícios

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:
   1. Implemente um *parser* para o reconhecimento de parênteses pareados. Veja o exemplo da gramática no slide 17 da aula 03.
   2. Modifique o exemplo 2 para reconhecer as quatro operações aritméticas básicas. As expressões podem ter parênteses. Veja exemplo no slide 10 da aula 03.
   3. Implemente um *parser* para o reconhecimento de comandos *if-then-else*. Veja o exemplo da gramática no slide 18 da aula 03.
   4. Modifique o exemplo 2 para realizar as operações de soma indicadas. Use uma variável global como acumulador.
   5. Crie um *parser* para reconhecer frases simples com a estrutura gramatical:

      ```
      artigo substantivo verbo adjetivo.
      ```

Laboratório 03 - Construindo *Parsers*

Prof. Eduardo Zambon

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

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