BISON

Prof. Lucas Mello Schnorr

Introdução

- `yacc` -Yet Another Compiler Compiler
- Produz um analisador ascendente para uma gramática
- Pode ser usado para produzir compiladores para várias linguagens
  - Pascal, C, C++, ...
- Outros usos
  - bc (calculadora)
  - eqn & pic (formatadores para troff)
  - Verificar a sintaxe SQL
  - Lex
- bison – versão livre da GNU

Especificação de Entrada

- Contém três seções
  - Definições (em C, incluído no início da saída)
  - Regras (Especificação da Gramática)
  - Código (em C, incluído no fim da saída)
- Sintaxe

```
Definições
%%
Regras
%%
Código Suplementar
```

Seção de Regras (seção principal)

- Contém a gramática
- Exemplo

```
expr : expr '+' term | term;
term : term '*' factor | factor;
factor : '(' expr ')'  | ID | NUM;
```

Seção de Definições (seção auxiliar)

- Contém a definição de tokens, símbolo inicial
- Exemplo

```
%{
  #include <stdio.h>
  #include <stdlib.h>
%}
%token ID NUM /* notar a declaração dos tokens */
%start expr
```

Interação com o analisador léxico

- Flex produz uma função `yylex()`
- Bison produz uma função `yyparse()`

- Flex e Bison: concebidos para interagirem
  - yyparse() chama yylex() para obter um token

- Duas opções
  - Ou implementamos manualmente yylex()
  - Ou utilizamos Flex diretamente (melhor opção)

Sequência básica operacional

- Supondo os arquivos
  - scanner.l com as especificações de tokens em lex
  - parser.y com a gramática em yacc

- Ordem de passos possível para construir o analisador

```
bison -d parser.y
flex scanner.l
gcc -c lex.yy.c parser.tab.c
gcc -o parser lex.yy.o parser.tab.o -lfl
```

- scanner.l deve incluir na seção de definições

```
#include "parser.tab.h"
```

Miscelânea

- As regras da gramática
  - Podem ser recursivas tanto a esquerda quanto a direita
  - Não podem ser ambíguas
- Usa um parser ascendente LALR(1)
  - Solicita um token
  - Empilha
  - Redução?
    - Sim $\rightarrow$ reduz usando a regra correspondente
    - Não $\rightarrow$ lê outro token na entrada
- Não pode olhar mais que um token de *lookahead*
- `bison -v parser.y` gera a tabela de estados

Exemplo (Lex) – arquivo `scanner.l`

```
%{
#include <stdio.h>
#include "parser.tab.h"
%}
id [_a-zA-Z][_a-zA-Z0-9]*
wspc [ \t\n]+
semi [;]
comma [,]
%%
int { return INT; }
char { return CHAR; }
float { return FLOAT; }
{comma} { return COMMA; }
{semi} { return SEMI; }
{id} { return ID;}
{wspc} {;}
```

Exemplo (Bison) – arquivo `parser.y`

```
%{
#include <stdio.h>
#include <stdlib.h>
%}
%start decl
%token CHAR COMMA FLOAT ID INT SEMI
%%
decl : type ID list;
list : COMMA ID list | SEMI;
type : INT | CHAR | FLOAT;
%%
```

Ações simples

Cada regra pode ter ações (código em C)
Exemplo

```
decl : type ID list { printf ("Sucesso!\n"); };
```

Ações e Atributos

- Cada regra pode ter ações (semânticas)
- Exemplo

```
E: E '+' E    { $$ = $1 + $3; }
 | INT_LIT    { $$ = INT_VAL; };
```

- `$n` é o atributo do n-ésimo símbolo na regra
- O default é que os atributos sejam do tipo inteiro
- Pode-se mudar o tipo através da diretiva

```
%token<...>  /* com o tipo do token */
%type<...>   /* tipo do não-terminal, com %union */
```

Ações e Atributos (Exemplo)

```
%union {
   char* nome;
   int inteiro;
   node* no;
}
%token<nome> IDF /* IDF terá atributo de tipo char* */
%type<no> E      /* E terá atributo de tipo node* */
%%
E: E  '+' E { $$ = create_node($1, $3, "plus"); }
| IDF        { $$ = create_leaf($1); };
```