Tradutores - 2º/2014
Laboratório 3 - *bison*

Luciano Santos
Profª Drª Cláudia Nalon

Instituto de Ciências Exatas
Departamento de Ciência da Computação
Universidade de Brasília

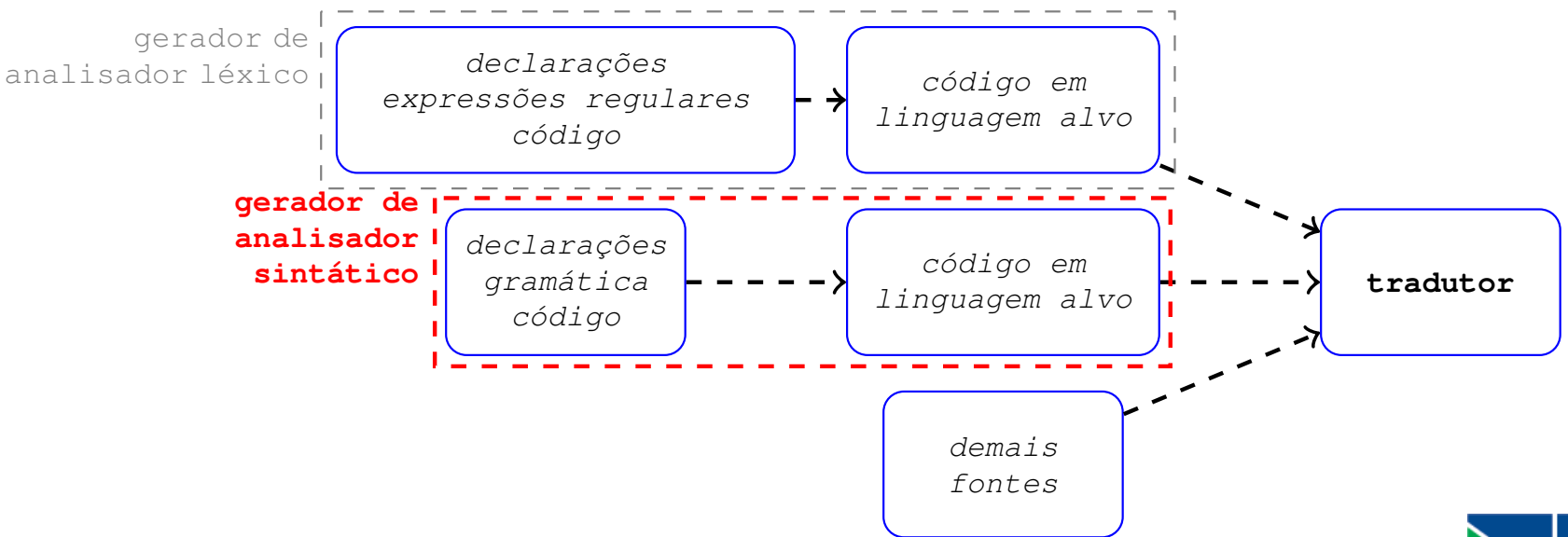
8 de outubro de 2014

Roteiro

- Geradores de analisadores sintáticos
- Estrutura de um programa *bison*
  - Esqueleto
  - Visão geral de cada seção
  - Regras sintáticas
  - Ações semânticas
- Exemplos
- Exercícios

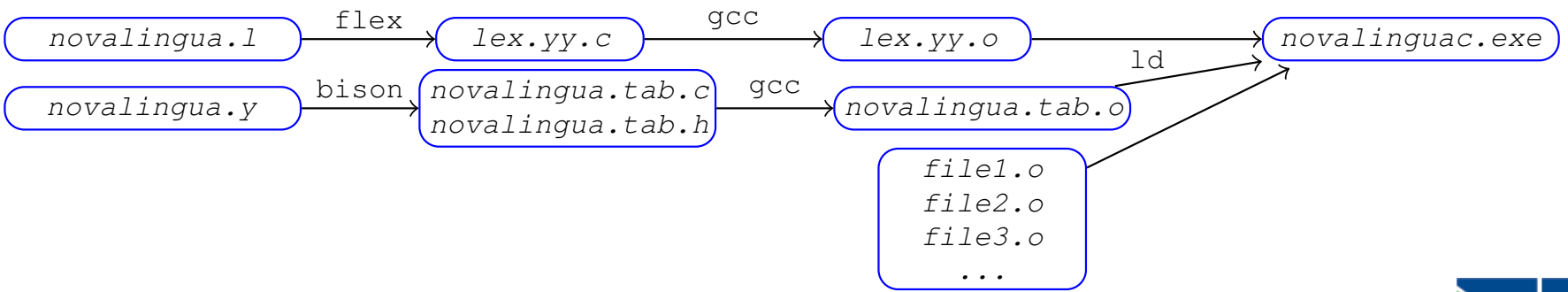
Geradores de Analisadores Sintáticos

- Um **gerador de analisadores sintáticos** recebe uma **especificação de linguagem (livre de contexto)**, geralmente na **BNF**, e gera um analisador sintático (também chamado de *parser*) em alguma linguagem de programação alvo, ou seja, gera os arquivos de código fonte que implementam o analisador:



Geradores de Analisadores Sintáticos

- Existem vários exemplos:
  - **yacc**: um dos mais utilizados, gera analisadores – $LALR(1)$, GLR, LR – em C;
  - **JavaCC**: gera analisadores léxico e sintático – $LL(k)$ – em Java;
  - **ANTLR**: gera analisadores – $LL(*)$ – em Java e C#.
- Vamos estudar o **bison**, uma implementação *open source* do *yacc* com algumas extensões.



Fluxo de utilização do *flex* e do *bison*.

Estrutura de um programa *bison*

```
%{
prólogo
%}
declarações


%%


regras sintáticas


%%


epílogo
```

Estrutura básica de um arquivo *bison*.

Estrutura de um programa *bison*

```
/* Prólogo (código no início do arquivo) */
/* Calculadora de notação polonesa. */
%{
  #include <stdio.h>
  #define YYSTYPE double

  /* geraremos esta função com o flex */
  extern int yylex();
  /* protótipo da função de tratamento de erros do bison */
  void yyerror (char const *);
%}

/* define um novo não terminal (token) chamado NUM */
%token NUM
```

Prólogo e declarações.

Estrutura de um programa *bison*

```
%% /* Regras Gramaticais */

/* aqui começa nossa gramática, uma palavra válida é zero ou mais linhas de
      entrada */
input:
  /* string vazia */
  | input line
;

/* linha é uma linha vazia ou uma expressão seguida de quebra de linha */
line:
  '\n'
  | exp '\n' { printf ("%.10g\n", $1); }
;

/* define recursivamente as possiveis expressões */
exp:
  NUM { $$ = $1; }
  | exp exp '+' { $$ = $1 + $2; }
  | exp exp '-' { $$ = $1 - $2; }
  | exp exp '*' { $$ = $1 * $2; }
  | exp exp '/' { $$ = $1 / $2; }
;
```

Regras gramaticais.

Estrutura de um programa *bison*

```
%%

/* Epílogo (código no final do arquivo) */

/* Chamado quando há um erro sintático na entrada. */
void yyerror (char const *s) {
  fprintf (stderr, "%s\n", s);
}

int main (void) {
  /* Chama o parser. */
  return yyparse ();
}
```

Epílogo.

Regras sintáticas

- Uma regra sintática é, em geral, da forma:
  `<resultado>: <componentes> ;`
- Onde:
  - ***resultado*** é o símbolo **não terminal** definido por esta regra;
  - ***componentes*** são sequências de **terminais** e **não terminais** que compõem esta regra, separadas pelo símbolo |.
- Por exemplo, o não terminal *expressao*, poderia ser definido como:

```
expressao: expressao '+' expressao
         | expressao '-' expressao;
```

Regras sintáticas

- O *bison*, por padrão, implementa um algoritmo **LALR(1)**, que aceita a linguagem definida pela gramática livre de contexto dada como entrada.
- O **símbolo inicial** é o primeiro que vier na seção de regras, ou aquele que for definido com a diretiva `%start` (veremos um exemplo).
- É importante notar que as gramáticas podem ser **ambíguas**. O *bison* fornece algumas ferramentas para identificar e remover ambiguidades, mas é papel do projetista criar uma gramática sem ambiguidades desde o princípio.
- Vamos estudar um exemplo...

Regras sintáticas

- Observe a gramática definida no arquivo `exemplo2.y`:

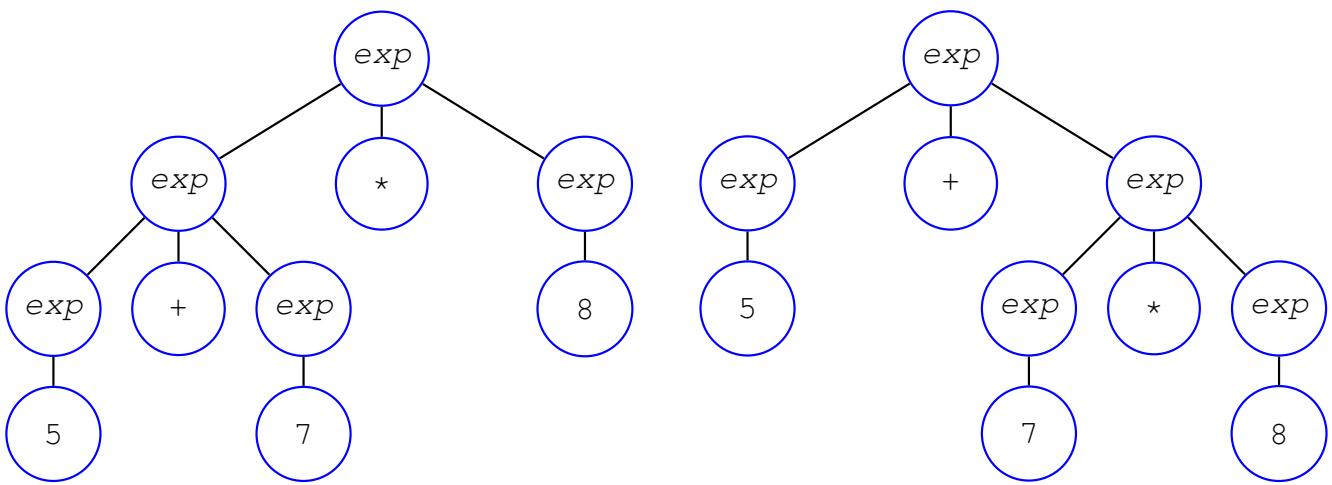
```
input:
  /* empty */
  | input line
;

line:
  '\n'
  | exp '\n'
;

exp:
  NUM
  | exp '+' exp
  | exp '-' exp
  | exp '*' exp
  | exp '/' exp
  | exp '^' exp
```

Regras sintáticas

- Invoque o *bison* para esta gramática, executando:
  `> bison exemplo2.y`
- Note que o *bison* encontrou conflitos do tipo *shift-reduce*.
- Isso aconteceu porque a gramática original é ambígua. Por exemplo, a entrada `5 + 7 * 8` deve gerar qual das duas árvores sintáticas abaixo?



Regras sintáticas

- Neste caso, a ambiguidade pode ser facilmente resolvida, de duas maneiras:
- Podemos alterar a gramática:

```
exp:
  parcel
  | exp '+' parcel
  | exp '-' parcel
;

parcel:
  factor
  | parcel '*' factor
  | parcel '/' factor
;

factor:
  NUM
  | NUM '^' factor
;
```

Regras sintáticas

- Ou podemos definir a **precedência de operadores** na seção de declarações do *bison*:

```
/* ... */

%token NUM
%left '-' '+'
%left '*' '/'
%right '^'

%%
```

Ações semânticas

- Após cada um dos símbolos no **corpo** da regra, pode ser inserida uma **ação semântica**, isto é, um código C a ser executado quando este símbolo é empilhado pelo algoritmo do bison.
- Dentro desta ação semântica, é possível **referenciar os símbolos na regra** utilizando a seguinte sintaxe:
  - `$$ = <valor>` define o valor do símbolo **resultado** desta regra;
  - `$n` referencia o **n-ésimo símbolo** na regra, começando por 1.
- Por exemplo:

```
exp:  NUM | exp '+' exp {$$ = $1 + $3};
```

- Esta regra referencia as duas instâncias de *exp*, pela ordem em que aparecem – ou seja, 1° e 3° símbolos da regra – soma os valores e atribui ao resultado.
- O tipo dos valores dos símbolos é definido pela macro `YYSTYPE`, mas pode ser sobrescrito em regras específicas, conforme veremos a seguir.

Árvore de Expressões

- Vamos escrever um programa que recebe como entrada uma expressão aritmética infixada, com os operadores +, −, / e ∗ e mostra na tela uma representação desta expressão em forma de árvore.
- Exemplo:

```
> 5 + 7 * 9 / 6
> +{
>     5
>     /{
>         *{
>             7
>             9
>         }
>         6
>     }
> }
```

Árvore de Expressões

- Primeiro, vamos utilizar o *flex* para identificar os tokens de nossa linguagem:

```
%{
  /* Este arquivo é gerado pelo bison, ele contém as definições de tipo e
        de constantes para os tokens. */
  #include "arvore.tab.h"
%}

%option noyywrap
%option bison-bridge

D    [0-9]
EOL "\n"|"\r"|"\r\n"
WS   [ \t\v\f]

%%

{D}+       { yylval->num = atoi(yytext); return NUM; }
"-"        { return '-'; }
"+"        { return '+'; }
"*"        { return '*'; }
"/"        { return '/'; }
{EOL}      { return '\n'; }
{WS}+      { BEGIN(INITIAL); }
.         { return *yytext; }

%%
```

Árvore de Expressões

```
%error-verbose // mostra os erros
%debug // ativa o modo de debug
%defines // cria o arquivo tab.h com as definições
%pure-parser // força o bison a criar um parser puro e definir a variável yylval

%code requires {
#include <stdlib.h>
#include <stdio.h>

/* Cria um novo tipo para o não terminal expression. */
typedef struct Expression Expression;
struct Expression {
  Expression *left;
  Expression *right;
  char op;
  int value;
};

/* Protótipos de funções. */
Expression* new_expr(Expression*, Expression*, char op, int);
void show_tree(Expression*, int);
void destroy_tree(Expression*);
void yyerror (char const *);
}
```

Árvore de Expressões - Arquivo *bison* – Prólogo.

Árvore de Expressões

```
/* Define o que os elementos na pilha será uma union. Os nomes dos campos da
      union serão os tipos dos símbolos. */
%union {
  int num;
  Expression *expr;
}

/* Informa ao bison que o valor do terminal NUM é "num" (campo da union) e que o
       nome desse token (para depuração) é "number". */
%token <num> NUM "number"

/* Informa ao bison que o valor do não terminal expression é "expr" (campo da
      union). */
%type <expr> expression

/* Precedência de operadores. */
%left '-' '+'
%left '*' '/'

/* Define o símbolo inicial da gramática. */
%start input

%%
```

Árvore de Expressões - Arquivo *bison* – Definição de tipos.

Árvore de Expressões

```
input: | input line;
line: '\n' | expression '\n' { printf("tree:\n"); show_tree($1, 0); destroy_tree
      ($1); printf("\n> "); } ;

expression:
  NUM { $$ = new_expr(0, 0, 0, $1); } // cria um novo nó apenas com o valor
  // cria nós para cada tipo de expressão
  | expression '+' expression { $$ = new_expr($1, $3, '+', 0); }
  | expression '-' expression { $$ = new_expr($1, $3, '-', 0); }
  | expression '*' expression { $$ = new_expr($1, $3, '*', 0); }
  | expression '/' expression { $$ = new_expr($1, $3, '/', 0); }
;

%%
```

Árvore de Expressões - Arquivo *bison* – Gramática.

Árvore de Expressões

```c
/* Aloca dinamicamente uma nova expressão com os valores dados. */
Expression* new_expr(Expression* l, Expression* r, char op, int value) {
  Expression *e = (Expression*) malloc(sizeof(Expression));
  e->left = l;
  e->right = r;
  e->op = op;
  e->value = value;
  return e;
}

/* Mostra a árvore de expressões na saída padrão. */
void show_tree(Expression *root, int tabs) {
  int i;
  for (i = 0; i < tabs; ++i) printf("  ");
  if (root->op) {
    printf("%c{\n", root->op);
    show_tree(root->left, tabs + 1);
    show_tree(root->right, tabs + 1);
    for (i = 0; i < tabs; ++i) printf("  ");
    printf("}\n");
  }
  else
    printf("%d\n", root->value);
}
```

Árvore de Expressões

```
/* Destrói a árvore de expressões alocada dinamicamente. */
void destroy_tree(Expression *root) {
  if (root->left) destroy_tree(root->left);
  if (root->right) destroy_tree(root->right);
  free(root);
}

void yyerror (char const *s) {
  fprintf (stderr, "%s\n", s);
}

int main (void) {
  return yyparse();
}
```

Árvore de Expressões - Arquivo *bison* – Funções II.

Árvore de Expressões

- Para testar este exemplo, execute na linha de comando:

```
> bison arvore.y
> flex arvore.l
> gcc -c -o lex.yy.o lex.yy.c
> gcc -c -o arvore.tab.o arvore.tab.c
> gcc -o arvore.exe lex.yy.o arvore.tab.o
```

- Teste seu analisador, fazendo:

```
> ./arvore
```

e digitando expressões válidas (e inválidas também).

Exercícios I

1. Amplie o programa de árvore de expressões para que aceite parênteses como uma forma de alterar a precedência de operadores. Duas dicas: uma expressão entre parênteses é equivalente a um número; você provavelmente só conseguirá resolver esse problema alterando a gramática.
2. Altere seus analisadores para aceitar como operandos nomes de identificadores com até 255 caracteres. Crie um novo tipo de comando que cria (e inicializa) um novo identificador. Crie uma tabela de símbolos para armazenar os identificadores.
3. Faça, utilizando o *bison*, uma nova versão da calculadora em notação polonesa estudada nos laboratórios de *flex* (incluindo as extensões implementadas nos exercícios).

Exercícios II

4. Implemente uma nova linguagem que aceite 3 tipos de comando:

   - `;` (comando vazio),
   - **`{`** *<zero ou mais comandos>* **`}`** (bloco de comandos),
   - **`if (`***<expr>***`)`** *<comando>* com bloco **`else`** opcional.

   O fato de que o bloco **`else`** é opcional gera uma ambiguidade, afinal no caso *ifs* aninhados, a qual *if* pertence um *else*? Imagine que a palavra chave `else` seja um operador, como podemos utilizar as diretivas `%left` ou `%right` para fazer com que o *else* sempre seja associado ao *if* mais próximo? Resolva esta ambiguidade!