Tradutores - 2º/2014
Laboratório 4 - *bison* mais avançado

Luciano Santos
Profª Drª Cláudia Nalon

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

9 de outubro de 2014

Roteiro

- Lidando com a localização dos *tokens*
- Resolvendo conflitos
  - Tipos de conflito
  - Visualizando o autômato do *bison*
  - Analisando o *log* do *bison*
- Exercícios

Lidando com a localização dos *tokens*

- Para reportar erros corretamente, é necessário controlar a **linha** e **coluna** atual na entrada.
- Esta tarefa dever ser realizada pelo **analisador léxico**, que repassa essa informação para as fases posteriores da tradução.
- O *bison*, quando trabalha em conjunto com o *flex* provê mecanismos para facilitar o armazenamento desta informação.

Lidando com a localização dos *tokens*

- Primeiro, é necessário adicionar a diretiva `%locations` à seção de declarações do *bison*. Esta opção **ativa o tratamento de localizações** de *tokens* pelo *bison* e muda a **assinatura** da função `yylex` para incluir um parâmetro adicional que receberá, por referência, as localizações de cada *token*.
- Em seguida, devemos incluir, além de `%bison-bridge`, também a opção `%bison-locations` no *flex*, que **declara o parâmetro** `yylloc` em `yylex`.

```
%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
      definir a variável yylval
%locations // ativa o tratamento de
      localizações

/* restante do arquivo... */
```

```
%{
  /* Inclui arquivo  de definições
        gerado pelo bison. */
  #include "arvore.tab.h"
%}

%option noyywrap
%option bison-bridge bison-locations

/* restante do arquivo... */
```

Lidando com a localização dos *tokens*

- Você pode criar seu próprio tipo personalizado para a variável `yylloc`, basta definir a macro `YYLTYPE` (de maneira similar à que fizemos com `YYSTYPE`).
- Por padrão, o *bison* já declara uma *struct* com campos inteiros para armazenar as posições de início e de final de cada *token*:

```
typedef struct YYLTYPE
{
  int first_line;
  int first_column;
  int last_line;
  int last_column;
} YYLTYPE;
```

- Para acessar a posição de um símbolo qualquer nas ações semânticas, utilize a notação `@n` para acessar o n-ésimo símbolo na regra ou `@$` para acessar localização da regra inteira.
- Vamos estudar um exemplo de como utilizar essa *struct* no *flex* (`locations.l` e `locations.y`).

Lidando com a localização dos *tokens*

- Gere e compile o exemplo:

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

- Teste seu analisador, fazendo:

```
> ./locations locations.in.txt
```

  e digitando expressões válidas (e inválidas também).

Tipos de conflito

- O *bison* pode encontrar 2 tipos de conflito em uma especificação de gramática, ambos relacionados a situações em que o autômato não é capaz de decidir a próxima ação a ser tomada:
  - *shift-reduce* – quando, para um estado qualquer, é possível tanto realizar um *shift* quanto aplicar uma regra de redução;
  - *reduce-reduce* – quando, para um estado qualquer, é possível aplicar mais de uma regra de redução para o mesmo símbolo;

Visualizando o autômato do *bison*

- A primeira ferramenta fornecida pelo *bison* para resolver problemas na gramática é uma representação visual do autômato. Para gerar esta representação, basta passar a opção `-g` para o *bison*.
- Teste esta opção com o arquivo de exemplo `expressao.y`:
  ```
  > bison -g expressao.y
  ```
- Note que além dos arquivos tradicionais, o *bison* também gera um arquivo `expressao.dot`. Esse arquivo está em formato `dot`, que pode ser manipulado pelo *Graphviz*[1]. Para converter para `pdf`, por exemplo, é possível fazer:
  ```
  > dot -Tpdf expressao.dot -o expressao.pdf
  ```

[1] http://www.graphviz.org/

Analisando o *log* do *bison*

- Uma maneira simples de entender em quais estados do autômato foram encontrados conflitos é analisar o *log* detalhado do *bison*, gerado quando ele recebe a opção `-v`.
- Execute:
  ```
  > bison -v expressao.y
  ```
- O *bison*, desta vez, gera o arquivo adicional `expressao.output`.
- Note que ele contém a **gramática**, os símbolos **terminais** e **não terminais**, os **estados** e, se houver, os conflitos em cada estado e a **ação padrão** adotada pelo *bison*.
- Em conflitos *shift-reduce*, a ação padrão é reduzir; em conflitos *reduce-reduce*, a ação padrão é reduzir pela regra que **vier primeiro**.
- Em qualquer situação, não confie nas ações padrão, resolva os conflitos!

Analisando o *log* do *bison*

- Analise agora o exemplo `primary.y`:
  `> bison -v primary.y`
- Este exemplo gera um conflito *reduce-reduce*. Neste exemplo, uma expressão primária em uma versão muito simples da linguagem C é definida como:
  - uma constante numérica;
  - um nome de variável;
  - um nome de um campo de enumeração.
- Observe que as duas últimas opções são ambíguas, porque ambas significam um único identificador. O comportamento padrão do *bison* seria sempre reduzir pela regra da enumeração, mas isto é um erro! E quando se tratar de uma variável?

Analisando o *log* do *bison*

- Para resolver este tipo de problema é necessário criar tipos diferentes de símbolos para cada caso.
- Uma possível solução seria, por exemplo, criar um novo tipo de *token* `ENUM_FIELD`, e o analisador léxico passa a buscar a tabela de símbolos toda vez que encontrar um identificador. Se encontrar o identificador em uma enumeração previamente definida, retorna o tipo correto, senão retorna `ID`.
- Nesse caso, um identificador só seria considerado um campo de enumeração se já houvesse sido definido, o que é um comportamento razoável na linguagem C.
- No entanto, para cada linguagem é necessário analisar o problema e resolver os conflitos!

Exercícios I

1. Crie um programa que lê um arquivo de entrada, informado via linha de comando, contendo uma ou mais expressões numéricas terminadas por ponto-e-vírgula, ignorando-se espaços em branco e quebras de linha. Seu programa deverá mostrar, para cada expressão, sua representação em forma de árvore, com as posições (linha e coluna) de início e de fim de cada nó da árvore ao lado da raiz. Observe o exemplo:

```
Entrada:  5 + 7 *\n9 / 6;
> + [1;1]-[2;5]{
>     5 [1;1]-[1;1]
>     / [1;5]-[2;5]{
>         * [1;5]-[2;1]{
>             7 [1;5]-[1;5]
>             9 [2;1]-[2;1]
>         }
>         6 [2;5]-[2;5]
>     }
> }
```

Exercícios II

2. Altere a gramática abaixo para resolver os conflitos *shift-reduce* e *reduce-reduce* no *bison*:

```
input:
     | input assign ;
assign:
     term
     | ID '=' assign ;
term:
     prim
     | term '+' prim
     | term '-' prim;
prim:
     ID
     | '(' term ')'
     | unary_op term
     | term unary_op ;
unary_op:
     OP_INC

     | OP_DEC ;
```