Compiladores - JACC

Fabio Mascarenhas - 2013.2

http://www.dcc.ufrj.br/~fabiom/comp

JACC

- Gerador de analisadores sintáticos LALR que gera código Java
- Sintaxe baseada na sintaxe do YACC (e de muitos outros geradores)
- Rápido, codifica o autômato LALR em código ao invés de usar uma tabela
- Usa o mesmo modelo de *ações semânticas* do YACC, o que permite gerar diretamente uma árvore sintática abstrata a partir da gramática

Usando o JACC

- Linha de comando: `jacc X.jacc`
  - -v: escreve saída do autômato em arquivo X.output
  - -h: escreve autômato em formato HTML no arquivo XMachine.html
  - -fv, -fh: mostra conjuntos FIRST e FOLLOW para cada não-terminal em conjunto com as opções anteriores
  - -a, -s, -0: usa estratégia de parsing LALR, SLR, ou LR(0)

Especificação JACC

- Formato do arquivo de entrada:

```
diretivas
%%
regras
%%
código extra
```

- Diretivas controlam a geração do parser
- Regras especificam a gramática e as ações durante a análise
- Código extra é inserido dentro da classe do parser

Regras

- As regras são onde especificamos as regras da gramática, separando as alternativas de cada não-terminal com :, e terminando com ;

```
expr : expr '+' term
     | expr '-' term
     | term
     ;

term : term '*' fact
     | term '/' fact
     | fact
     ;

fact : '(' expr ')'
     | NUM
     ;
```

- Terminais (tokens) são caracteres entre '' ou nomes em maiúsculas

Ações de redução

- Cada alternativa pode ter código Java associado que é executado na redução, e pode dizer qual valor deve ser empilhado para aquela redução

- O valor retirado na pilha para cada símbolo da alternativa fica em pseudo-variáveis $1, $2, ..., e o valor da pseudo-variável $$ é empilhado

```
expr : expr '+' term     { $$ = new ast.Soma($1, $3); }
     | expr '-' term     { $$ = new ast.Sub($1, $3); }
     | term              // default { $$ = $1; }
     ;

term : term '*' fact     { $$ = new ast.Mul($1, $3); }
     | term '/' fact     { $$ = new ast.Div($1, $3); }
     | fact
     ;

fact : '(' expr ')'      { $$ = $2; }
     | NUM               { $$ = new ast.Num($1.val); }
     ;
```

Precedência e associatividade

- A gramática pode ser ambígua na parte dos operadores, usando as diretivas %left e %right para resolver a ambiguidade dando a precedência e associatividade dos mesmos

- Precedência é declarada da menor para a maior

```
%left '+' '-'
%left '*' '/'
%right '^'

%%

expr : expr '+' expr  { $$ = new ast.Soma($1, $3); }
     | expr '-' expr  { $$ = new ast.Sub($1, $3); }
     | expr '*' expr  { $$ = new ast.Mul($1, $3); }
     | expr '/' expr  { $$ = new ast.Div($1, $3); }
     | expr '^' expr  { $$ = new ast.Exp($1, $3); }
     | NUM            { $$ = new ast.Num($1.val); }
     ;
```

Operadores unários e binários

- Se um operador tem duas precedências, uma quando é unário e outra quando binário, deve-se usar um pseudo-token e a diretiva %prec na regra:

```
%left '+' '-'
%left '*' '/'
%right '^'
%left NEG

%%

expr : expr '+' expr   { $$ = new ast.Soma($1, $3); }
     | expr '-' expr   { $$ = new ast.Sub($1, $3); }
     | expr '*' expr   { $$ = new ast.Mul($1, $3); }
     | expr '/' expr   { $$ = new ast.Div($1, $3); }
     | expr '^' expr   { $$ = new ast.Exp($1, $3); }
     | '-' expr %prec NEG { $$ = new ast.Neg($2); }
     | NUM
     ;
```

Diretivas

- Vários aspectos da configuração do analisador gerado
- %package e %class dão o pacote e o nome da classe do analisador
- %implements diz quais interfaces o analisador deve implementar
- %interface dá o nome da interface gerada com as definições de tokens
- %next dá o nome do método chamado para obter o próximo token, e %get o nome do campo que contém o código do token de lookahead
- %semantic dá o tipo e nome do campo que contém o token de lookahead

Mais diretivas

- %{ e %} permitem embutir código Java antes da classe (imports, por exemplo)
- Diretivas %token dão os nomes de todos os tipos de token que não são dados por um único caractere
- Diretivas %type dão os tipos dos valores semânticos de cada não-terminal da gramática

```
%type <Comando> cmd
%type <Expressao> exp
%type <Lvalue> lval
%type <List> exps cmds
```

Código extra

- Deve-se sempre declarar um método yyerror para tratamento de erros sintáticos:

```
void yyerror(String msg) {
   ...
}
```

- Também é preciso fazer a adaptação entre a interface do analisador léxico e o que o JACC espera, implementando o método dado na diretiva %next e os campos das diretivas %get e %semantic

- Um bug no JACC faz o método parse() gerado não retornar o valor semântico do programa como um todo, então também precisa-se usar um campo auxiliar para isso