Appendix 3 A Grammar for Grammars

The grammar for CUP

We can also write a grammar to describe the structure of a grammar. The parser for CUP programs is itself written using CUP. (Obviously the first version of CUP was not written this way, but later version could be). With the actions and the ":label"s deleted, the grammar for CUP is in fact:

```
package java_cup;
import java_cup.runtime.*;
import java.util.Hashtable;

action code {:...:};

parser code {:...:};

init with {:...:};
scan with {:...:};

terminal
     PACKAGE, IMPORT, CODE, ACTION, PARSER, TERMINAL, NON, INIT, SCAN, WITH,
     START,  SEMI,  COMMA,  STAR,  DOT,  COLON,  COLON_COLON_EQUALS,  BAR,
PRECEDENCE,
     LEFT, RIGHT, NONASSOC, PERCENT_PREC, LBRACK, RBRACK, NONTERMINAL;
terminal String
     ID, CODE_STRING;
non terminal
     spec, package_spec, import_list, action_code_part,
     code_parts, code_part, opt_semi, non_terminal,
     parser_code_part, symbol_list, start_spec, production_list,
     multipart_id, import_spec, import_id, init_code, scan_code, symbol,
     type_id,      term_name_list,      non_term_name_list,      production,
prod_part_list,
     prod_part, new_term_id, new_non_term_id, rhs_list, rhs, empty,
     precedence_list, preced, terminal_list, precedence_l, declares_term,
     declares_non_term;
non terminal String
     nt_id, symbol_id, label_id, opt_label, terminal_id, term_id, robust_id;

start with spec;

spec ::=
          package_spec
          import_list
          code_parts
          symbol_list
          precedence_list
          start_spec
          production_list
     |
          error
          symbol_list
          precedence_list
          start_spec
          production_list
     ;

package_spec ::=
          PACKAGE multipart_id SEMI
```

```
     |
          empty
     ;

import_list ::=
          import_list import_spec
     |
          empty
     ;

import_spec ::=
          IMPORT import_id SEMI
     ;

code_part ::=
          action_code_part
     |
          parser_code_part
     |
          init_code
     |
          scan_code
     ;

code_parts ::=
     |
          code_parts code_part
     ;

action_code_part ::=
          ACTION CODE CODE_STRING_code opt_semi
     ;

parser_code_part ::=
          PARSER CODE CODE_STRING_code opt_semi
     ;

init_code ::=
          INIT WITH CODE_STRING_code opt_semi
     ;

scan_code ::=
          SCAN WITH CODE_STRING_code opt_semi
     ;

symbol_list ::=
          symbol_list symbol
     |
          symbol
     ;

symbol ::=
          TERMINAL type_id declares_term
     |
          TERMINAL declares_term
     |
          non_terminal type_id declares_non_term
     |
          non_terminal declares_non_term
     |
          TERMINAL error SEMI
```

```
     |
          non_terminal error SEMI
     ;

declares_term ::=
          term_name_list SEMI
     ;

declares_non_term ::=
          non_term_name_list SEMI
     ;

term_name_list ::=
          term_name_list COMMA new_term_id
     |
          new_term_id
     ;

non_term_name_list ::=
          non_term_name_list COMMA new_non_term_id
     |
          new_non_term_id
     ;

precedence_list ::=
          precedence_l
     |
          empty
     ;

precedence_l ::=
          precedence_l preced
     |
          preced
     ;

preced ::=
          PRECEDENCE LEFT terminal_list SEMI
     |
          PRECEDENCE RIGHT terminal_list SEMI
     |
          PRECEDENCE NONASSOC terminal_list SEMI
     ;

terminal_list ::=
          terminal_list COMMA terminal_id
     |
          terminal_id
     ;

terminal_id ::=
          term_id
     ;


term_id ::=
          symbol_id
     ;

start_spec ::=
          START WITH nt_id_name SEMI
```

```
     |
          empty
     ;

production_list ::=
          production_list production
     |
          production
     ;

production ::=
          nt_id_id COLON_COLON_EQUALS rhs_list SEMI
     |
          error SEMI
     ;

rhs_list ::=
          rhs_list BAR rhs
     |
          rhs;

rhs ::=
          prod_part_list PERCENT_PREC term_id_name
     |
          prod_part_list

     ;

prod_part_list ::=
          prod_part_list prod_part
     |
          empty
     ;

prod_part ::=
          symbol_id opt_label
     |
          CODE_STRING_str
     ;

opt_label ::=
          COLON label_id
     |
          empty
     ;

multipart_id ::=
          multipart_id DOT robust_id_id
     |
          robust_id_id
     ;

import_id ::=
          multipart_id DOT STAR
     |
          multipart_id
     ;

type_id ::=
          multipart_id
     |
```

```
          type_id LBRACK RBRACK
     ;

new_term_id ::=
          ID_id
     ;

new_non_term_id ::=
          ID_term_id
     ;

nt_id ::=
          ID_id
     |
          error
     ;

symbol_id ::=
          ID_id
     |
          error
     ;

label_id ::=
          robust_id_id
     ;

robust_id ::=
          ID_id
     |
          CODE
     |
          ACTION
     |
          PARSER
     |
          TERMINAL
     |
          NON
     |
          NONTERMINAL
     |
          INIT
     |
          SCAN
     |
          WITH
     |
          START
     |
          PRECEDENCE
     |
          LEFT
     |
          RIGHT
     |
          NONASSOC
     |
          error
     ;
```

```
non_terminal ::=
          NON TERMINAL
     |
          NONTERMINAL
     ;

opt_semi ::=
     |
          SEMI
     ;

empty ::=
     ;
```

An alternative grammar for grammars

I do not like the CUP grammar definition, so here is my own grammar definition for a grammar definition.

The lexical Analyser

At the lexical level, identifiers are terminal symbols, whether they represent terminals or nonterminals in the grammar they are defining.

```
import java.io.*;
import java_cup.runtime.*;

%%

%public
%type         java_cup.runtime.Symbol
%char

%{
     int currentLine = 1;
     int lineStart = 0;

     public java_cup.runtime.Symbol createToken( int tokenType ) {
          return new java_cup.runtime.Symbol(
               tokenType, yychar, yychar + yytext().length(), yytext() );
          }

     public java_cup.runtime.Symbol createToken( int tokenType, Object value
) {
          return new java_cup.runtime.Symbol(
               tokenType, yychar, yychar + yytext().length(), value );
          }
%}

%init{
     yybegin( NORMAL );
%init}

%eofval{
     return createToken( sym.EOF );
%eofval}

%state NORMAL LEXERROR
%%
<NORMAL>"terminal"              { return createToken( sym.TERMINAL ); }
<NORMAL>"nonterminal"           { return createToken( sym.NONTERMINAL ); }
<NORMAL>"start"                 { return createToken( sym.START ); }
```

```
<NORMAL>"::="                    { return createToken( sym.EXPANDSTO ); }
<NORMAL>"|"                      { return createToken( sym.OR ); }
<NORMAL>","                      { return createToken( sym.COMMA ); }
<NORMAL>";"                      { return createToken( sym.SEMICOLON ); }
<NORMAL>[A-Za-z][A-Za-z0-9]*     {
                                 return createToken( sym.IDENT, yytext() );
                                 }
<NORMAL>[\ \t]                   { }
<NORMAL>[\n]                     { currentLine++; }
<NORMAL>.                        {
                                 System.out.println( "Got error character "
                                      + ( int ) yytext().charAt( 0 ) );
                                 yybegin( LEXERROR );
                                 return createToken( sym.error );
                                 }
<LEXERROR>.*\n                   {
                                 currentLine++;
                                 yybegin( NORMAL );
                                 }
```

The parser

The grammar for a grammar is fairly straightforward. As I process the list of terminals, nonterminals, and rules, I add them to a global database.

```
import java.util.*;
import java_cup.runtime.*;

parser code
     {:

     Yylex lexer;

     public parser( Yylex lexer ) {
          this.lexer = lexer;
          }

     public void report_error( String message, Object info ) {
          System.err.print( message );
          if ( info != null && info instanceof java_cup.runtime.Symbol ) {
               java_cup.runtime.Symbol  token  =  (  java_cup.runtime.Symbol  )
info;
               System.err.println( " ... at line "
                    + lexer.currentLine + ":"
                    + ( token.left - lexer.lineStart ) );
               System.err.println( " ... with value " + token.value );
               }
          else
               System.err.println();
          }

     :};

scan with
     {:
     java_cup.runtime.Symbol token = lexer.yylex();
     return token;
     :};

/* Terminal Tokens */
terminal
     TERMINAL,
     NONTERMINAL,
```

```
     COMMA,
     START,
     EXPANDSTO,
     OR,
     SEMICOLON;

terminal  String
     IDENT;

nonterminal Grammar;
nonterminal TerminalDecl;
nonterminal TerminalList;
nonterminal NonterminalDecl;
nonterminal NonterminalList;
nonterminal StartDecl;
nonterminal RuleList;
nonterminal Rule;
nonterminal Vector RhsList;
nonterminal Vector Rhs;

start with Grammar;

Grammar::=
               TerminalDecl
               NonterminalDecl
               StartDecl
               RuleList
          ;

TerminalDecl::=
               TERMINAL
               TerminalList
               SEMICOLON
          |
               error
          ;

TerminalList::=
               IDENT:name
               {:
               new Terminal( name );
               :}
          |
               TerminalList
               COMMA
               IDENT:name
               {:
               new Terminal( name );
               :}
          |
               error
          ;

NonterminalDecl::=
               NONTERMINAL
               NonterminalList
               SEMICOLON
          |
               error
          ;
```

```
NonterminalList::=
               IDENT:name
               {:
               new Nonterminal( name );
               :}
          |
               NonterminalList
               COMMA
               IDENT:name
               {:
               new Nonterminal( name );
               :}
          |
               error
          ;

StartDecl::=
               START
               IDENT:name
               SEMICOLON
               {:
               Symbol startSymbol = Symbol.get( name );
               if ( startSymbol == null )
                    throw new Error( name + " undeclared" );
               else if ( ! ( startSymbol instanceof Nonterminal ) )
                    throw new Error( "Start symbol "
                         + name + " must be nonterminal" );
               else {
                    Nonterminal.setStartSymbol(  (  Nonterminal  )  startSymbol
);
                    }
               :}
          |
               error
          ;

RuleList::=
          |
               RuleList
               Rule
          ;

Rule::=
               IDENT:name
               EXPANDSTO
               RhsList:rhsList
               SEMICOLON
               {:
               Symbol symbol = Symbol.get( name );
               if ( symbol == null )
                    throw new Error( name + " undeclared" );
               else if ( ! ( symbol instanceof Nonterminal ) )
                    throw new Error( name + " must be nonterminal" );
               else {
                    Nonterminal lhs = ( Nonterminal ) symbol;
                    lhs.addRules( rhsList );
                    }
               :}
          |
               error
               SEMICOLON
```

```
          ;

RhsList::=
               Rhs:rhs
               {:
               Vector rhsList = new Vector();
               rhsList.addElement( rhs );
               RESULT = rhsList;
               :}
          |
               RhsList:rhsList
               OR
               Rhs:rhs
               {:
               rhsList.addElement( rhs );
               RESULT = rhsList;
               :}
          ;

Rhs::=
               {:
               Vector rhs = new Vector();
               RESULT = rhs;
               :}
          |
               Rhs:rhs
               IDENT:name
               {:
               Symbol symbol = Symbol.get( name );
               if ( symbol == null )
                    throw new Error( name + " undeclared" );
               rhs.addElement( symbol );
               RESULT = rhs;
               :}
          ;
```