

Delphi Parser Generator user's guide

October 29, 2002

Contents

1	Introduction	1 - 1
1.1	Overview	1 - 2
1.2	Features	1 - 2
1.3	Installation	1 - 3
2	Getting started	2 - 1
2.1	Lexical analyzer	2 - 2
2.2	Parser	2 - 3
2.3	The project	2 - 5
3	Syntactic elements	3 - 1
3.1	General	3 - 3
3.1.1	Comments	3 - 3
3.1.2	White Space	3 - 3
3.1.3	Symbols	3 - 3
3.2	Atomic production elements	3 - 4
3.2.1	Character literal	3 - 4
3.2.2	String literal	3 - 4
3.2.3	Wildcard	3 - 4
3.2.4	Token reference	3 - 4
3.3	Simple production elements	3 - 5
3.3.1	Rule reference	3 - 5
3.3.2	Semantic action	3 - 6
3.4	Production element operators	3 - 7
3.4.1	Element complement	3 - 7

3.4.2	Set complement	3 - 7
3.4.3	Range operator	3 - 7
3.4.4	Ignore operator	3 - 7
3.5	Sections	3 - 8
3.5.1	unit	3 - 8
3.5.2	uses	3 - 8
3.5.3	const	3 - 8
3.5.4	type	3 - 8
3.5.5	options	3 - 9
3.5.6	tokens	3 - 9
3.5.7	memberdecl	3 - 9
3.5.8	memberdef	3 - 10
3.5.9	parser	3 - 10
3.5.10	lexer	3 - 11
3.5.11	rule definitions	3 - 11
3.6	Options	3 - 13
3.6.1	k	3 - 13
3.6.2	importVocab	3 - 14
3.6.3	exportVocab	3 - 14
3.6.4	testLiterals	3 - 15
3.6.5	caseSensitive	3 - 16
3.6.6	filter	3 - 16
3.6.7	ignore	3 - 17
3.7	Element labels	3 - 18
3.8	EBNF rule elements	3 - 18
3.9	Rule arguments	3 - 18
3.10	Exception handlers	3 - 18
3.10.1	Exception handler for a rule	3 - 18
3.10.2	Exception handler for an alternative	3 - 19
3.10.3	Default error handler in lexer	3 - 19

4	Grammars	4 - 1
4.1	Structure of a grammar	4 - 2
4.1.1	Unit declaration	4 - 2
4.1.2	Unit sections	4 - 2
4.1.3	Grammar class definition	4 - 2
4.1.4	Grammar class sections	4 - 3
5	Tokens	5 - 1
5.1	Overview	5 - 2
5.2	Defining tokens	5 - 2
5.3	User defined token classes	5 - 5
6	Run-time	6 - 1
6.1	Error handling	6 - 2
6.1.1	DPG exception hierarchy	6 - 2
6.1.2	EdpgException	6 - 2
6.1.3	EdpgMismatchedChar	6 - 2
6.1.4	EdpgMismatchedToken	6 - 3
6.1.5	EdpgSemantic	6 - 3
A	Grammar of Delphi Parser Generator	A - 1
A.1	Lexical analyzer	A - 2
A.2	Parser	A - 8

1

Introduction

Contents

1.1	Overview	1-2
1.2	Features	1-2
1.3	Installation	1-3

1.1 Overview

The Delphi Parser Generator is a language tool which automatically generates $LL(k)$ parsers in Object Pascal Language based on an intuitive grammar, similar to EBNF. The generated code mimics a hand-written parser, so that it is easier to debug and leads to shortened development time compared to state-machine based LR or DFA/NFA parsers. To compensate theoretical limitations of $LL(k)$ parsers, DPG features several powerful extensions enhancing its functionality far beyond that of standard $LL(k)$ parsers. The method of syntactic and semantic predicates makes the writing of meta-parsers simple and routine. The philosophy of DPG is to allow the programmer maximum control over the parsing process while eliminating all the routine work.

1.2 Features

- Delphi code generator for $LL(k)$ lexers and parsers.
- Intuitive and consistent EBNF like syntax for both the lexer and the parser generator resulting in a shallow learning curve.
- Extremely easy-to-read generated code undistinguishable from hand-written parsers. The inlined statements are properly indented relative to the surrounding program code.
- Syntactic predicates allow for conditional parsing based on formal syntactic conditions, enhancing the functionality of the $LL(k)$ parsers considerably.
- Semantic predicates allow for conditional parsing based on essentially arbitrary conditions. For example, a DOM-based XML parser is easily written by semantic predicates using an internal hash-table representation of the DOM. Using traditional state-machine based parsers (like YACC), programmers often need to delegate parsing tasks to the hand written part of the code. This burdens them with laborious and error-prone routine work. Semantic predicates prevent this, since the parser is allowed to use run-time information for the parsing process dynamically.
- Actions can be inserted in the rules at every possible place. These actions can be used for controlling the parsing process with high granularity.
- All rules may have return values and arguments. Rule arguments add a powerful metaparsing capability completing the predicate and action mechanism optimally.
- All rules may have a code initialization section. This special feature is tuned for Pascal to allow the programmer to declare and initialize local variables for each rule.

- Many convenient extensions to the plain BNF syntax, such as (\dots) , $(\dots)?$, $(\dots)+$, $(\dots)^*$, which simplify the task of writing grammars and makes it less error-prone.
- Element complements allow for matching a text not matching a given rule.
- Element labels are used to directly map rule information to Pascal variables. They provide a seamless interaction between the generated and user-written code.
- Intuitive Graphical User Interface with syntax highlighting, and project management capabilities.

1.3 Installation

The first step in using DPG is to install it in Delphi. However, before using DPG be sure to read over the License Agreement.

- run setup.exe and follow the instructions
- run Delphi and add your DPG run-time library directory to Delphi's library path. For example, to do this for Delphi 6 select *Tools | Environment Options* on the menu bar. Go to the *Library* tab and add the full path of your DPG run-time directory to the *Library Path* if you have not already done so.

2

Getting started

Contents

2.1	Lexical analyzer	2-2
2.2	Parser	2-3
2.3	The project	2-5

In this chapter, we develop a simple calculator. It accepts integers, the four arithmetic operators (+, -, /, *), and parenthesis on its input. Spaces, tabs and newline characters are treated as white spaces and used for separating tokens. Complete Expressions must be terminated by semicolons.

2.1 Lexical analyzer

Let us define the calculator's lexer.

```
1 unit myLexer;  
2  
3 lexer TmyLexer;  
4 options  
5 {  
6   exportvocab = myLexer;  
7 }
```

In line 1 we define the unit name of the generated Pascal source file for the lexer. In line 3 we give a name to the lexer class. If there is an `options` block for a grammar class, it must follow the class declaration. Here, we define one option for the lexer: `exportVocab`. This option tells the DPG that all the token definitions must be exported to `myLexerTokens.txt` and `myLexerTokens.pas`. Grammars can import the generated token names using the exported `.txt` files.

Note: it is not necessary to define the `exportVocab` option for a grammar. The file names for the token exchange files are automatically created using the specified unit name.

Now we define the lexer tokens.

```
8 LPAREN: '(' ;  
9 RPAREN: ')' ;  
10 PLUS: '+' ;  
11 MINUS: '-' ;  
12 STAR: '*' ;  
13 SLASH: '/' ;  
14 SEMI: ';' ;
```

In lines from 8 to 14, there are simple token definitions. Each of them recognizes one character from the input stream.

```
15 INT: ('0'..'9')+ ;
```

In line 15, we define a rule to recognize integer numbers. This tells us that the INT consists of one or more numeric characters.

Now, define a rule to handle white space characters.

```
16 WS
17   : '\r' '\n'      { _ttype := TT_SKIP; }
18   | '\t'           { _ttype := TT_SKIP; }
19   | ' '            { _ttype := TT_SKIP; }
20   ;
```

Characters surrounded by curly braces are actions. The content of an action block will be copied verbatim into the generated Pascal source file. In this example the expression `_ttype := TT_SKIP;` forbids the WS rule to generate a token, because we don't need it.

Now the lexer definition is finished. This simple lexer recognizes relevant characters, integers and skips every white spaces on its input.

2.2 Parser

Now we define the parser.

```
1 unit myParser;
2
3 parser TmyParser;
4 options
5 {
6   importVocab = myLexer;
7 }
```

This part is analogous to lexer definition with one exception. In line 6, we import the tokens from a file specified by the `exportVocab` option in the lexer grammar. Now, the parser knows which tokens are to be expected from the lexer.

```
8 memberdecl
9 {
10   value: integer;
11 }
```

In lines from 8 to 11, we specify the `memberdecl` section. This section is used to define members for the generated parser class. In this example, the `TmyParser` class will have a member called `value`. We use this member to store the result of the calculation for the current expression.

Now we define the rules.

```
12 calc
13   : (expression SEMI { writeln( value); } ) *
14   ;
15
16 expression
17 local
18 {
19   temp : integer;
20 }
21   : term                { temp := value;          }
22   (
23       PLUS term        { temp := temp + value;    }
24   |   MINUS term       { temp := temp - value;    }
25   ) *                  { value := temp;          }
26   ;
```

In lines 17..20, we define a local variable for the rule expression. The following rules are defined in a similar way to the rule expression.

```
27 term
28 local
29 {
30   temp : integer;
31 }
32   : factor              { temp := value;          }
33   (
34       STAR factor       { temp := temp * value;   }
35   |   SLASH factor      { temp := temp div value; }
36   ) *                  { value := temp;          }
37   ;
38
39 factor
40 local
41 {
42   temp : integer;
43 }
44   : uInt
45   | LPAREN expression RPAREN
46   ;
47
48 uInt
49   : x:INT { value := StrToInt( x.TokenText); }
50   ;
```

In line 49, we specified that the rule must have a variable called 'x' which will contain the INT token. For the moment, it is enough to know that it has a property `TokenText` which contains the text of the recognized token. This property is a string property, so we have to convert it to an integer, and store it in the `value` member variable.

2.3 The project

The following simple project demonstrates how the defined lexer and parser classes are used.

```
1 program calc;
2 {$APPTYPE CONSOLE}
3 uses
4   Classes,
5   SysUtils,
6   myLexer in 'myLexer.pas',
7   myParser in 'myParser.pas';
8
9 var
10  stm: TFileStream;
11  lex: TmyLexer;
12  par: TmyParser;
13
14 begin
15   if ParamCount <> 1 then
16     begin
17       writeln('usage: calc <filename>');
18       exit;
19     end
20   else
21     begin
22       try
23         stm := TFileStream.Create( ParamStr(1),
24                                   fmOpenRead);
25         lex := TmyLexer.Create(stm);
26         par := TmyParser.Create(lex);
27
28         par.calc;
29       except
30         on EdpgMismatchedToken do
31           writeln('Syntax error');
32         on EdpgMismatchedChar do
33           writeln('Syntax rrror');
34       end;
35     end;
36
37   stm.Free;
38   par.Free;
39 end.
```


3

Syntactic elements

Contents

3.1	General	3-3
3.1.1	Comments	3-3
3.1.2	White Space	3-3
3.1.3	Symbols	3-3
3.2	Atomic production elements	3-4
3.2.1	Character literal	3-4
3.2.2	String literal	3-4
3.2.3	Wildcard	3-4
3.2.4	Token reference	3-4
3.3	Simple production elements	3-5
3.3.1	Rule reference	3-5
3.3.2	Semantic action	3-6
3.4	Production element operators	3-7
3.4.1	Element complement	3-7
3.4.2	Set complement	3-7
3.4.3	Range operator	3-7
3.4.4	Ignore operator	3-7
3.5	Sections	3-8
3.5.1	unit	3-8
3.5.2	uses	3-8
3.5.3	const	3-8

3.5.4	type	3 - 8
3.5.5	options	3 - 9
3.5.6	tokens	3 - 9
3.5.7	memberdecl	3 - 9
3.5.8	memberdef	3 - 10
3.5.9	parser	3 - 10
3.5.10	lexer	3 - 11
3.5.11	rule definitions	3 - 11
3.6	Options	3 - 13
3.6.1	k	3 - 13
3.6.2	importVocab	3 - 14
3.6.3	exportVocab	3 - 14
3.6.4	testLiterals	3 - 15
3.6.5	caseSensitive	3 - 16
3.6.6	filter	3 - 16
3.6.7	ignore	3 - 17
3.7	Element labels	3 - 18
3.8	EBNF rule elements	3 - 18
3.9	Rule arguments	3 - 18
3.10	Exception handlers	3 - 18
3.10.1	Exception handler for a rule	3 - 18
3.10.2	Exception handler for an alternative	3 - 19
3.10.3	Default error handler in lexer	3 - 19

Delphi Parser Generator (DPG) uses the ASCII character set, including the letters *A* through *Z* and *a* through *z*, the digits *0* through *9*, and other standard characters. It is case sensitive. The space character (ASCII 32), the tab character (ASCII 9), and the new-line characters (ASCII 13,10) are called *white-space* characters.

3.1 General

3.1.1 Comments

DPG accepts single and multi-line comments. Single-line comments begin with `//` while multi-line (block) comments are enclosed by `(* and *)`.

3.1.2 White Space

Spaces, tabs, and new-lines (including most used CR-LF, CR, LF constructions) are separators in that they separate DPG symbols, such as identifiers. White spaces have no additional significance i.e. the code layout does not play any semantical role. However the layout of the embedded Delphi code is preserved in the generated source files.

3.1.3 Symbols

DPG uses the following punctuation and keywords:

<code>(...)</code>	subrule	<code>unit</code>	unit name
<code>(...) *</code>	closure subrule	<code>uses</code>	uses section
<code>(...) +</code>	positive closure	<code>const</code>	const section
<code>(...) ?</code>	optional subrule	<code>type</code>	type section
<code>[...]</code>	rule arguments	<code>lexer</code>	lexer class
<code>{ ... }</code>	semantic action	<code>parser</code>	parser class
<code>{ ... } ?</code>	semantic predicate	<code>options</code>	options section
<code>(...) =></code>	syntactic predicate	<code>tokens</code>	tokens section
<code> </code>	alternative operator	<code>returns</code>	rule return value
<code>..</code>	range operator	<code>except</code>	exception handler
<code>~</code>	not operator	<code>finally</code>	exception handler
<code>!</code>	ignore operator	<code>memberdecl</code>	member declaration
<code>.</code>	wildcard	<code>memberdef</code>	member definition
<code>=</code>	assignment operator	<code>local</code>	local rule variables
<code>:</code>	label, start rule		
<code>;</code>	end rule		

Table 3.1: DPG symbols

3.2 Atomic production elements

3.2.1 Character literal

Single characters enclosed in quotes are character literals. A character literal can only be referred to within a lexer rule. For example, ' { ' needs not be escaped as you are specifying the literal character which is to be matched. Meta symbols are used outside of characters and string literals to specify lexical structure. Special characters can be specified in a similar way to C escape sequences. DPG accepts the following escape sequences: \n, \r, \t, \', \", \\. The #xx form is not accepted by DPG.

3.2.2 String literal

String literals are sequences of characters enclosed in double quotes. The same escape sequences can be used in string literals as in character literals. In parser rules, strings represent tokens, and each unique string is assigned to a token type. Referring to a string within a lexer rule matches the indicated sequence of characters and is a shorthand notation. For example, consider the following equivalent lexer rule definitions:

```
BEGIN : "begin";  
BEGIN : 'b' 'e' 'g' 'i' 'n';
```

3.2.3 Wildcard

The wildcard . within a parser rule matches any single token; within a lexer rule it matches any single character.

3.2.4 Token reference

Identifiers beginning with an uppercase letter are treated as token references. The subsequent characters may be a mixture of letters, digits or underscores. Referencing a token in a parser rule implies that you want to recognize a token with the specified token type. This does not actually call the associated lexer rule – the lexical analysis phase delivers a stream of tokens to the parser. A token reference within a lexer rule implies a method call to that rule, and carries the same analysis semantics as a rule reference within a parser. So, you may specify rule arguments and return values for non-public tokens and for every parser rule. See the next section on rule references.

3.3 Simple production elements

3.3.1 Rule reference

Identifiers beginning with lowercase letter are treated as parser rule references. The subsequent characters may be any letter, digit, number, or underscore. Lexical rules may not reference parser rules. Referencing a rule implies a method call to that rule at that point in the parse. You may pass parameters and obtain return values. For example, formal and actual parameters are specified within square brackets:

```
function
  : type ID LPAREN args RPAREN block [1]
  ;

block [scope: integer]
  : LCURLY
    ...
    { (* use arg 'scope' *) }
    ...
  RCURLY
  ;
```

Return values that are stored in variables use a simple assignment notation:

```
set
local
{
  ids : TStringList;
}
{
  ids := nil;
}
: LPAREN ids=idList RPAREN
;

idList returns [TStringList]
{
  result := TStringList.Create;
}
: id:ID { result.Add( id.TokenText); }
(
  COMMA id:ID
  {
    result.Add( id.TokenText);
  }
)*
;
```

3.3.2 Semantic action

Actions are blocks of Object Pascal source code enclosed in curly braces. The code is executed after the preceding production element has been recognized and before the recognition of the following element. Actions are typically used to generate output, construct trees, or modify a symbol table. An action's position dictates when it is recognized relative to the surrounding grammar elements.

If the action is the first element of a production, it is executed before any other element in that production, but only if that production is predicted by the lookahead.

The first action of an EBNF subrule may be followed by `:`. Doing so designates the action as an init-action and associates it with the subrule as a whole, instead of any production. It is executed immediately upon entering the subrule, and is executed even while guessing (testing syntactic predicates). For example:

```
( { init-action} :  
  { action of 1st production} production1  
  | { action of 2nd production} production2  
 )?
```

The init-action would be executed regardless of what (if anything) matched in the optional subrule.

3.4 Production element operators

3.4.1 Element complement

The unary not operator \sim may be applied to an atomic element such as a token identifier. For some token atom T , $\sim T$ matches any token other than T except end-of-file. Within lexer rules, $\sim 'a'$ matches any character other than character $'a'$. The sequence $\sim .$ (“not anything”) is meaningless and not allowed. Example:

```
SL_COMMENT : "//" (~'\n')* '\n';
```

3.4.2 Set complement

The unary not operator \sim can also be used to construct a token set or character set by complementing another set. This is most useful when you want to match tokens or characters until a certain delimiter set is encountered. Rather than invent a special syntax for such sets, DPG allows the placement of \sim in front of a subrule containing only simple elements and no actions. The simple elements may be token references, token ranges, character literals, or character ranges. For example:

```
SL_COMMENT : "//" (~('\'r' | '\n'))* (\'r' | '\n');
```

3.4.3 Range operator

The binary range operator $..$ is used to define a range of atom which may be matched. The expression $c1..c2$ in a lexer matches characters included in that range. The expression $T..U$ in a parser matches any token whose token type is inclusively in that range, which is of dubious value if the token types are generated externally.

3.4.4 Ignore operator

In lexer grammars, the ignore operator $!$ can be applied to any atomic production element. It means that the element followed by the $!$ operator should not appear in the result token. Example:

```
STRING : "' '! (~'\"'')* '\"' ! ;
```

3.5 Sections

3.5.1 unit

The `unit` section specifies the unit name of the generated source file. The syntax is identical to Object Pascal.

3.5.2 uses

The `uses{...}` section is used to specify the units which must be included in the interface's `uses` clause of the generated pascal unit. Every unit name must be terminated by a semicolon. Repeated units are included only once.

```
uses
{
    Classes;
    Windows;
}
```

3.5.3 const

The `const{...}` section is used to specify items that appear in the interface's `const` clause of the generated pascal unit. The content of this section is copied verbatim into the unit.

```
const
{
    const1 = 12;
    const2 = FOO;
}
```

3.5.4 type

The `type{...}` section is used to specify items that appear in the interface's `type` clause of the generated pascal unit. The content of this section is copied verbatim into the unit.

```
type
{
    TmyType1 = integer;
    TmyType2 = array [0..16] of TmyType1;
}
```


3.5.5 options

The `options{...}` section contains options for a given grammar element. Options can be defined for lexer/parser classes, rules and subrules.

3.5.6 tokens

If you need to define an “imaginary” token (i.e. one that has no corresponding real input symbol) use the `tokens{...}` section to define them. You can also define literals in this section.

```
tokens
{
    "procedure";
    "function";
    INTEGER;
}
```

Strings defined in this way are treated just as if you had referenced them in the parser. The formal syntax is:

```
tokenSpecification
: "tokens"
  LCURLY
  (tokenItem SEMI)*
  RCURLY
;

tokenItem
: TOKEN
| STRING
;
```

The `tokens{...}` section is only valid in lexer grammars.

3.5.7 memberdecl

The `memberdecl{...}` section contains additional member declarations for the grammar class. It allows the expansion of the grammar class with user defined members, so it is not necessary to derive new classes from the generated class to implement additional functionality. The content of this section is copied verbatim into the class declaration of the generated grammar class.

```
memberdecl
{
    procedure proc1;
    procedure proc2;
}
```

3.5.8 memberdef

The `memberdef { . . . }` section contains the implementation of the classes' additional functionality. The content of this section is copied verbatim into the implementation part of the generated unit. This section may also contain the initialization and finalization clauses.

```
memberdef
{
    procedure TmyClass.proc1;
    begin
        ...
    end;

    procedure TmyClass.proc2;
    begin
        ...
    end;
}
```

3.5.9 parser

Parser rules must be associated with a parser class. Each parser class specification precedes the options, and rule definitions of the parser. Grammar files `.g` can hold only one class definition. A parser specification in a grammar file looks like:

```
unit myParser;
uses...           // optional uses {...} section
const...          // optional const {...} section
type...           // optional type {...} section

parser TmyParser;

options...        // optional options {...} section
memberdecl...     // optional memberdecl {...} section
parser rules...
memberdef...      // optional memberdef {...} section
```

In the generated code, the parser class results in an Object Pascal class, and the rules become member methods of the class.

Note, that the content of the `memberdecl{...}` section is copied verbatim into the class declaration part of the generated parser class while the content of the `memberdef{...}` section is copied after the implementation of the member rules, so the initialization and finalization clauses of a pascal unit can be placed in the `memberdef{...}` section.

3.5.10 lexer

To perform lexical analysis, you need to specify a lexer class that describes how to break up the input character stream into a stream of tokens. The syntax is similar to that of a parser class:

```
unit myLexer;  
uses...           // optional uses {...} section  
const...          // optional const {...} section  
type...           // optional type {...} section  
  
lexer TmyLexer;  
  
options...        // optional options {...} section  
tokens...         // optional tokens {...} section  
memberdecl...     // optional memberdecl {...} section  
lexer rules...  
memberdef...      // optional memberdef {...} section
```

Lexical rules contained within a lexer class become member methods in the generated class. A lexer grammar may have a `tokens{...}` section to specify imaginary tokens and string literals.

3.5.11 rule definitions

The structure of an input stream of atoms is specified by a set of mutually-referenced rules. Each rule has a name and any of the following optional attributes: a scope specifier; a set of arguments; an init-action; a return value; local variable definitions; an exception handler and an alternative or alternatives. Each alternative contains a series of elements that specify what to match and where. Scope can be specified by `private`, `protected`, or `public` keywords. A rule has public scope by default. The basic form of a rule is:

```
(scope) rulename  
: alternative_1  
| alternative_2  
...  
| alternative_n  
;
```

Syntactic elements

Parameters for a rule can be specified in the following form:

```
rulename [formal parameters] : ... ;
```

If the rule returns a value, its type can be defined with the `returns` keyword:

```
rulename returns [typename] : ... ;
```

where `typename` is a valid Object Pascal type specifier.

Local variables for a rule can be defined in the `local { ... }` section:

```
rule
local
{
    foo: integer;
    bar: string;
}
```

Init-actions are specified before the colon. Init-actions differ from normal actions because they are always executed regardless of guess mode.

```
rule
{
    init-action
}
: ... ;
```

Parser rules apply structure to a stream of tokens, whereas lexer rules apply structure to a stream of characters. Parser rules, therefore, must not reference character literals. Double-quoted strings in parser rules are considered to be token references. Note: all parser rules must begin with a lowercase letter.

Lexer rules defined within a lexer grammar must have a name beginning with an uppercase letter. These rules implicitly match characters on the input stream instead of tokens on the token stream. Referenced grammar elements include token references (implicit lexer rule references), characters and strings. Lexer rules are processed in the same manner as parser rules, and may also specify arguments and return values. A scope specifier for a lexer rule has special meaning in lexer grammars. In the generated Object Pascal unit, the lexer class has a `nextToken` function which is the interface between the lexer and the parser. This function is synthesized from the public lexer rules. It means that non-public lexer rules don't modify the prediction logic of the lexer. They are usually helper rules. If the lexer grammar has no public rule at all, the `nextToken` function returns EOF to the parser.

3.6 Options

The `options{...}` section is used to specify options for grammar elements. i.e. elements are the lexer/parser classes, rules and subrules. This section is preceded by the `options` keyword and contains a series of option/value assignments surrounded by curly braces.

3.6.1 `k`

synopsis: set lookahead depth
context: parser/lexer class declaration
type: integer
default: 1

For any grammar, the lookahead depth can be specified by using the k option.

```
lexer myLexer;  
options  
{  
    k = 2;  
}
```

Setting the lookahead depth changes the maximum number of tokens that will be examined to select alternative productions, and test for exit conditions of the EBNF constructs $(...)?$, $(...)+$, and $(...)*$. The lookahead analysis is linear approximate (as opposed to full $LL(k)$). Consider this example with $k = 2$:

```
r : ( A B | B A )  
    | A A  
    ;
```

Full $LL(k)$ analysis would resolve the ambiguity and produce a lookahead test for the first alternative like:

```
if (LA(1)=A and LA(2)=B) or (LA(1)=B and LA(2)=A)
```

Linear approximate analysis would logically OR the lookahead sets at each depth, resulting in a test like:

```
if (LA(1)=A or LA(1)=B) and (LA(2)=A or LA(2)=B)
```

Which is ambiguous for the second alternative for $\{A, A\}$. Therefore, setting the lookahead depth very high tends to yield diminishing returns in most cases, because the lookahead sets at large depths will include almost everything. This problem can be solved using a syntactic predicate.

3.6.2 importVocab

synopsis: set initial grammar vocabulary
context: parser/lexer class declaration
type: ID
default: none

The import vocabulary for a grammar class can be specified using the `importVocab` option.

```
lexer myLexer;  
options  
{  
    importVocab = XML;  
}
```

DPG will look for the token exchange file named `XMLTokens.txt`, and import all the token definitions from it. Parser grammar must use this option, because without that, it cannot communicate with the lexer. Lexer grammar can use this option too. It is useful, when a parser class uses multiple lexers to get tokens from the input stream. The vocabulary file has an identifier on the first line that names the token vocabulary. All subsequent lines are of the form `ID=value` or `ID="literal"=value`. For example:

```
ThocLexer  
TT_EOF = 1  
TT_LPAREN = 4  
TT_RPAREN = 5  
LT_const = "const" = 6
```

The token exchange file is automatically generated by DPG for each grammar.

Note: you must take care of the order of grammars in a DPG project. Vocabulary-generating grammars must appear before vocabulary-consuming grammars.

3.6.3 exportVocab

synopsis: set export grammar vocabulary
context: parser/lexer class declaration
type: ID
default: grammar class name

The vocabulary of a grammar is the union of the set of tokens provided by an `importVocab` option and the set of tokens and literals defined in the grammar.

```
lexer myParser;  
options  
{  
    exportVocab = XML1;  
}
```

If the `exportVocab` options isn't specified, then DPG will use the grammar class name to export the vocabulary. DPG generates the following files for the example above: `XML1Tokens.txt` for token exchange, and `XML1Tokens.pas` for the grammar class.

3.6.4 testLiterals

context: lexer class declaration, lexer rule
type: boolean
default: false

By default, DPG doesn't generate code to check the literals table (the table generated for literal strings), because checking the literals table after each token recognition is expensive. Instead, it checks string literals in a lexer rule, that can recognize them. The string literals table contains the strings defined in the `tokens{...}` section of a lexer grammar.

```
lexer myLexer;  
options  
{  
    testLiterals = false;  
}  
tokens  
{  
    "function";  
    "procedure";  
    ...  
}  
  
ID  
options  
{  
    testLiterals = true;  
}  
:  
    (A..Z | a..z)(A..Z | a..z | 0..9)*  
;
```

In the example above, if the input is matched by the rule `ID` then the implementation of the rule will check the literals table for the matched token. If it exists, then

the returned token type will be set to the token type assigned to the string literal in the literals table. Otherwise the returned token type will remain unchanged.

It is possible to check the literals table explicitly within an action using the `TestLiteral` method:

```
{
    ...
    _ttype := TestLiteral;
    _ttype := TestLiteral( _ttype);
    ...
}
```

3.6.5 caseSensitive

context: lexer class declaration
type: boolean
default: false

```
lexer myLexer;
options
{
    caseSensitive = true;
}
```

Case is ignored when comparing against character and string literals in the lexer. The case of the input stream is maintained when stored in the token objects.

3.6.6 filter

context: lexer class declaration
type: boolean / ID
default: false

```
lexer myLexer;
options
{
    filter = true;
}
```

When `true`, the lexer ignores any input not exactly matching one of the public lexer rules.

Notice that the filter rule must track new-lines in the general case where the lexer might emit error messages.

When set to a rule name, the filter rule is invoked either when the lookahead (in `nextToken`) predicts none of the public lexical rules or when one of those rules fails. In the latter case, the input is rolled back before attempting the filter rule. Option `filter=true` is like having a filter rule such as:

```
IGNORE : . ;
```

3.6.7 ignore

context: lexer rule
type: ID
default: none

```
lexer myLexer;  
options  
{  
    ignore = MyIgnoreRule;  
}
```

Specify a lexer rule to use a white space between lexical rule atomic elements (chars, strings, and rule references). The grammar analysis, and hence the lookahead sets, are aware of the whitespace references.

3.7 Element labels

Any atomic production element can be labeled by an identifier (case is insignificant). For a labelled atomic element, the identifier is used within a semantic action to access the associated Token object or character. For example,

```
assign
  : v:ID EQUALS expr SEMI
  {
    writeln(Assign to  + v.TokenText);
  }
;
```

3.8 EBNF rule elements

DPG supports the following extended BNF notations:

- (. . .) – exactly one occurrence of a subrule
- (. . .) ? – zero or one occurrence of a subrule
- (. . .) + – one or more occurrence of a subrule
- (. . .) * – zero or more occurrence of a subrule

3.9 Rule arguments

Character sequences in square brackets are arguments or return type specifiers. Square brackets within string and character literals are not argument delimiters. The arguments within [] must follow the Object Pascal syntax.

3.10 Exception handlers

DPG allows the specification of exception handlers specific to a given rule or alternative. The general form of an exception handler specification is:

```
... except { code to handle exception }
... finally { code to handle exception }
```

3.10.1 Exception handler for a rule

The exception handler for a rule must be placed after the terminating semicolon. The handler can be either an `except` block or a `finally` block. The implementation of a rule will be surrounded by a `try` block.

```
r : ...  
  ;  
  except { handler code }
```

3.10.2 Exception handler for an alternative

The exception handler of an alternative must be the last element of the alternative. Both exception handler blocks can be used. Every alternative that has an exception block will be surrounded by a `try...except/finally` block.

```
r : alternative_1 ... except { handler code }  
  | alternative_2 ... finally { handler code }  
  ...  
  | alternative_n  
  ;
```

Note: It is not necessary to define an exception handler for each alternative.

3.10.3 Default error handler in lexer

To skip every character that isn't recognized by any public lexer rule, specify the option `filter=true` for a lexer. That way, the parser doesn't have to deal with lexical errors and ask for another token.

4

Grammars

Contents

4.1	Structure of a grammar	4-2
4.1.1	Unit declaration	4-2
4.1.2	Unit sections	4-2
4.1.3	Grammar class definition	4-2
4.1.4	Grammar class sections	4-3

4.1 Structure of a grammar

The generic structure of a DPG grammar is the following:

- *unit declaration*
- *unit sections*
- *grammar class definition*
- *grammar class sections*

Note: the order of blocks cannot be changed!

4.1.1 Unit declaration

The *unit declaration* is always the first block in any DPG grammar. It specifies the name of the target Pascal unit generated by DPG from the grammar. The syntax is identical to that of Delphi.

```
unit UnitName ;
```

4.1.2 Unit sections

The *unit sections* block must follow the *unit declaration* block if it exists. The members of this block are optional, but they must appear in the following order:

- *uses section*
- *const section*
- *type section*

4.1.3 Grammar class definition

This block defines the type of the grammar class. The possible types are `lexer` and `parser`.

```
lexer myLexer ; // define lexer
```

or

```
parser myParser ; // define parser
```

4.1.4 Grammar class sections

This block may contain the following sections in the order specified:

- *options section*
- *tokens section* (only for lexers)
- *memberdecl section*
- *rule definitions*
- *memberdef section*

5

Tokens

Contents

5.1	Overview	5-2
5.2	Defining tokens	5-2
5.3	User defined token classes	5-5

5.1 Overview

Tokens are the basic building blocks of any parser or compiler. The task of a lexer (lexical analyzer, scanner) is to break up the input character stream into a stream of tokens. Then `nextToken` method of a lexer passes the next token to the parser, or throws an exception if the next character on the input stream cannot be matched by any of the public lexer rules. The `nextToken` method is always synthesized from the public lexer rules.

Tokens in DPG are interface pointers. The interface type is `IdpgToken`, which defines the following properties:

```
IdpgToken = interface
...
property TokenText    : string;
property TokenType    : integer;
property TokenLine    : integer;
property TokenColumn  : integer;
...
end;
```

where `TokenText` is the text matched by the lexer; `TokenType` is the type of token assigned to the token by DPG; `TokenLine` is the line number where the token starts in the input stream; `TokenColumn` is the column number.

Within parser rules, the input token can be accessed via this interface. To obtain the interface to the recognized token, the reference to the token must be prefixed by a label. For example,

```
...
x:NUMBER
{
...
  LogMsg( 'Token: ' + x.TokenText );
  LogMsg( 'Type:   ' + IntToStr(x.TokenType));
...
}
...
```

Note: Variables for labels are always generated by DPG, so you should not define them in the `local{...}` section of the rule.

5.2 Defining tokens

In DPG, tokens can be defined in the lexer grammars. DPG always generates a token exchange file that describes all the token types matched by the lexer. This

file can be imported in a parser grammar, so the lexer and parser have the same token types. Tokens can be defined either,

- via lexer rules, or
- in the lexer's `tokens{...}` section

Defining a token using a lexer rule

The commonest method of defining a token is using a lexer rule. In lexer grammars, every rule is associated with a `TokenType` which is determined by DPG at compile time. This value is assigned to the result token by default, but it can be modified in the given rule if needed. This is used mostly in rules that need runtime information to set the type of the result token, but is otherwise uncommon.

There is one exception: when a rule must not generate a token at all. This is useful for defining comments or white-spaces for a grammar. Every lexer rule has a local variable called `_ttype`. If `_ttype` has a value of `TT_SKIP`, then the rule won't generate any token. For example,

```
SLCOMMENT : "//" ( ~'\n' )* '\n' { _ttype := TT_SKIP; } ;
```

The following examples are normal lexer rules, and they are typical in lexers:

```
LPAREN:      '(' ;
RPAREN:      ')' ;
DIGIT:       '0'..'9' ;
NUMBER:      DIGIT (DIGIT)* ;
LETTER:      'a'..'z' | 'A'..'Z' ;
ID:          LETTER (LETTER | DIGIT | '_' )* ;
```

Defining a token in the `tokens{...}` section

Lexer grammars may have a `tokens{...}` section in the class declaration. Within this section you can define “imaginary” tokens and string literals. These tokens are not “real” tokens and cannot be referenced in lexer rules. “Imaginary” tokens are helpful when a rule can recognize more than one type of token and defining rules for these tokens would be ambiguous. For example,

```
tokens
{
    STRING;
    CHAR;
}
```

Tokens

```
// =====  
// String or char  
// =====  
STRING_OR_CHAR  
:  '\'' (~'\'' | '\'' '\'')* '\''  
  {  
      if TokenText = ' ' then _ttype := TT_STRING  
      else if TokenText = ' ' then _ttype := TT_CHAR  
      else if Length( TokenText) > 3 then _ttype := TT_STRING  
      else _ttype := TT_CHAR;  
  }  
;
```

The rule `STRING_OR_CHAR` recognizes a pascal character literal, and a pascal string literal. The code in the action block decides which type of token must be created by the rule. Note: These tokens are “imaginary” tokens. Referencing them in lexer grammars is not possible, because they have no implementation. Within parser rules, the tokens `STRING` and `CHAR` can be referenced. But `STRING_OR_CHAR` can’t be referenced, because this rule creates a `STRING` or a `CHAR` token.

String literals in the `tokens{...}` section are useful when the language defines keywords. In this case you can list your language’s keywords in this section. They will be put into the lexer’s literals table. The lexer will consult this table in the following cases:

- if the `testLiterals` option for the lexer class is true, the lexer checks the literals table after each recognized token,
- if the `testLiterals` option for the lexer class is false, the check will be executed in rules, that have this option set.

If neither lexer rules nor lexer class have this option set, the lexer’s literals table can be explicitly checked via the `TestLiterals` method. The advantage of using string literals is that you can reference them in the parser as they are defined in the `tokens{...}` section. For example,

```
...  
lexer TmyLexer;  
options  
{  
    testLiterals = true;  
}  
tokens  
{
```

```
...
"function";
"procedure";
...
}
...

parser TmyParser;
rule1 : "function" ID SEMI;
rule2 : "procedure" ID LPAREN args RPAREN SEMI;
...
```

In the above example we set the `testLiterals` option to true for the lexer class. This is not recommended, because the lexer will check the literals table even if it found a non-string token. Instead, you have to check the table in a rule that can recognize these literals. For example:

```
...
lexer TmyLexer;
...

ID
options
{
    testLiterals=true;
}
: 'a'..'z' | 'A'..'Z' ('a'..'z' | 'A'..'Z' | '0'..'9')*
;
```

Here the literals table will only be consulted in the rule `ID`. This will improve the lexer's speed. Of course you can set the `testLiterals` options to true for as many rules as you want. All of them will check the literals table.

Note: The `testLiterals` option has no effect for lexer rules.

5.3 User defined token classes

By default, DPG uses the `TdpgToken` class to represent tokens. This class is derived from `TInterfacedObject`, and implements the `IdpgToken` interface. This interface is used across the generated code. To define a new token class you must derive your new class from `TdpgToken`, implement your interface to access and manipulate your object, and finally tell the lexer that it must create your type of token object instead of the default `TdpgToken`. After that, within the rules you

must obtain the interface of your class and use it. Let us have a more detailed look at this:

1. Create a token class:

```
ImyToken = interface( IdpgToken)
  [a guid definition]

  function   Get_MyString : string;
  procedure Set_MyString( AString: string);

  property  MyString : string    read   Get_MyString
                                     write Set_MyString;
end;

TmyToken = class( TdpgToken,
                  IdpgToken,
                  ImyToken)
protected
  fMyString : string;

  function   Get_MyString : string;
  procedure Set_MyString( AString: string);

public
  constructor Create(  pType: integer;
                      pText: string); override;

end;

constructor TmyToken.Create( pType: integer;
                           pText: string);
begin
  inherited;
  ...
  your code here
  ...
end;

function TmyToken.Get_MyString: string;
begin
  result := fMyString;
end;

function TmyToken.Set_MyString( pString: string);
begin
  fMyString := pString;
end;
```

2. Tell to lexer that it must use our token class.

```
uses myToken;  
...  
myLexer.TokenClass := TmyToken;
```

3. Use it in a rule.

```
...  
parser TmyParser;  
  
rule1  
:  
    "procedure" x:id (LPAREN params RPAREN)?  
    {  
        (x as TmyToken).MyString := 'procid';  
    }  
;
```

Note: You must cast the returned interface to your token interface, because the `makeToken` method of the lexer always returns an `IdpgToken` interface and the labels specified to obtain a reference to a token are always `IdpgToken` references.

Note: If you have to do special actions to initialize your token class, you must have the same constructor as defined in the example. The `makeToken` method of the lexer always creates tokens with this constructor. If you have another kind of constructor for your token class, it won't be used by the lexer.

6

Run-time

Contents

6.1	Error handling	6-2
6.1.1	DPG exception hierarchy	6-2
6.1.2	EdpgException	6-2
6.1.3	EdpgMismatchedChar	6-2
6.1.4	EdpgMismatchedToken	6-3
6.1.5	EdpgSemantic	6-3

6.1 Error handling

All syntactic and semantic errors throw exceptions. In particular, the methods used to match tokens in the parser base class (match etc) throw `EdpgMismatchedToken`. The methods in the lexer base class used to match characters (match etc) throw exceptions similarly.

6.1.1 DPG exception hierarchy

DPG-generated parsers throw exceptions to signal recognition errors or other stream problems. All exceptions derive from `EdpgException`. The hierarchy is as follows:

```
EdpgException
  EdpgMismatchedChar
  EdpgMismatchedToken
  EdpgSemantic
```

6.1.2 EdpgException

The `EdpgException` is the base class for all DPG exceptions. It defines the following read-only properties:

```
FileName : string;
Line     : integer;
Column   : integer;
```

These properties contain information about the location where the exception occurred.

6.1.3 EdpgMismatchedChar

The `EdpgMismatchedChar` exception is thrown by the lexer when it is looking for a character, but finds a different one on the input stream than expected. It defines the following properties in addition to those of `EdpgException`.

```
FoundChar    : char;
FoundString   : string;
CharSet       : TdpgCharSet;
Str           : string;
Inverted      : boolean;
```

The `FoundChar` and `FoundString` properties contain the character or string that was found on the input stream. The `CharSet` and `Str` properties contain the values which the lexer expected to find. The `Inverted` property is set only if the exception came from a `MatchNot(...)` operation. In this case, the `CharSet` property contains the values, that the lexer must not match. The validity of properties are shown in the next table, depending on the kind of exception.

	Mismatched char	Mismatched string
<code>FoundChar</code>	valid	-
<code>FoundString</code>	-	valid
<code>CharSet</code>	valid	-
<code>Str</code>	-	valid
<code>Inverted</code>	valid	-

6.1.4 `EdpgMismatchedToken`

The `EdpgMismatchedToken` exception is thrown by the parser when it is looking for a token, but finds a different one on the input token stream than expected. It defines the following properties in addition to those of `EdpgException`.

```
FoundToken    : IdpgToken;  
TokenSet      : TdpgByteSet;  
Inverted      : boolean;
```

The `FoundToken` property contains the token the parser received from the lexer. The `TokenSet` property contains the values the parser expected to get. The `Inverted` property is set only if the exception came from a `MatchNot(...)` operation. In this case, the `TokenSet` property contains the values the parser must not get.

6.1.5 `EdpgSemantic`

This exception is thrown by a validating semantic predicate. It defines the following property in addition to those of `EdpgException`.

```
Assert : string;
```

The `Assert` property contains the validating expression that caused the exception.



Grammar of Delphi Parser Generator

A.1 Lexical analyzer

```
unit dpgDpgLexer;  
  
lexer TdpgDpgLexer;  
options  
{  
    testLiterals = false;  
    k             = 2;  
}  
  
tokens  
{  
    "unit";  
    "uses";  
    "const";  
    "type";  
  
    "lexer";  
    "parser";  
  
    "options";  
    "tokens";  
    "memberdecl";  
    "memberdef";  
  
    "private";  
    "protected";  
    "public";  
  
    "returns";  
    "local";  
  
    "except";  
    "finally";  
  
    SEMPRED;  
  
    USES;  
    OPTIONS;  
    TOKENS;  
}  
  
// -----  
// Simple tokens  
// -----  
LPAREN:      '(' ;  
RPAREN:      ')' ;  
RCURLY:      '}' ;
```

```
COLON:      ':' ;
SEMI:       ';' ;
COMMA:      ',' ;
ASSIGN:     '=' ;
IMPLIES:    '>=' ;
QUEST:      '?' ;
PLUS:       '+' ;
STAR:       '*' ;
NOT:        '~' ;
OR:         '|' ;
BANG:       '!' ;
WILDCARD:   '.' ;
RANGE:      '..' ;
```

```
// -----
// Character literal
// -----
CHARLIT
:  '\''! (ESC | ~'\'' ) '\''! ;

// -----
// String literal
// -----
STRINGLIT
:  '"' (ESC | ~'"')* '"' ;

// -----
// Integer
// -----
INTEGER local
{
  i: integer;
  v: integer;
}
:  DNUMBER
{
  v := 0;
  for i:=1 to Length( TokenText) do
  begin
    v := v * 10 + ord( TokenText[i]) - ord('0');
  end;

  TokenText := IntToStr( v);
}
;
```

```
// -----
// Argument action
// -----
ARGACTION
:
    '[' !
    (
        '\r' '\n' { newLine; }
        | '\r'      { newLine; }
        | '\n'      { newLine; }
        | ~',' '
    ) *
    ']' !
;

// -----
// Action
// -----
ACTION
:
    '{'
    (
        '\r' '\n' { newLine; }
        | '\r'      { newLine; }
        | '\n'      { newLine; }
        | ~',' '
    ) *
    '}'
    ( '?' ! { _ttype := TT_SEMPRED; } ) ?
;

// -----
// Token ref
// -----
TOKENREF
options
{
    testLiterals = true;
}
: 'A'..'Z' ('a'..'z' | 'A'..'Z' | '_' | '0'..'9')* ;

// -----
// Rule ref
// -----
RULEREF
local
{
    t: integer;
}

```



```
:
    t = INT_RULEREF { _ttype := t; }
    (
        {t = LT_uses}?      WS_LOOP ( '{' { _ttype := TT_USES; } )?
        | {t = LT_options}? WS_LOOP ( '{' { _ttype := TT_OPTIONS; } )?
        | {t = LT_tokens}?  WS_LOOP ( '{' { _ttype := TT_TOKENS; } )?
    )?
;

// -----
// Internal rule ref
// -----
protected INT_RULEREF returns [integer]
{
    _ttype := TT_RULEREF;
}
: 'a'..'z' ('a'..'z' | 'A'..'Z' | '_' | '0'..'9')*
{
    result := TestLiteral( _ttype);
}
;

// -----
// COMMENT
// -----
COMMENT
: SLCOMMENT { _ttype := TT_SKIP; }
| MLCOMMENT { _ttype := TT_SKIP; }
;

// -----
// SLCOMMENT
// -----
protected SLCOMMENT
:
    " //"
    ( ~( '\r' | '\n' ) )*
    (
        '\r' '\n' { newLine; }
        | '\r'      { newLine; }
        | '\n'      { newLine; }
    )
;


```

```
// -----
// Multi line comment version
// Nested comments aren't allowed!
// -----
protected MLCOMMENT
:
    " ( *"
    (
        options
        {
            greedy = false;
        }
        : '\r' '\n' { newLine; }
        | '\r'      { newLine; }
        | '\n'      { newLine; }
        | .
    ) *
    "*" ) "
;

// -----
// Numbers
// -----
protected DNUMBER: '0'..'9' (DDIGIT)*;
protected DDIGIT: '0'..'9';

// -----
// WS
// -----
WS
:
    (
        ' '
        | '\t' { tab; }
        | '\r' '\n' { newLine; }
        | '\r' { newLine; }
        | '\n' { newLine; }
    )
    {
        _ttype := TT_SKIP;
    }
;


```

```
// -----  
// WS_LOOP  
// -----  
protected  
WS_LOOP  
:  
  (  
    options  
    {  
      greedy = true;  
    }  
    : WS  
    | COMMENT  
  )*  
;  
  
// -----  
// Esc  
// -----  
protected  
ESC  
:  
  '\\\\'! ( 'r' | 'n' | 't' | '\\'' | '\"' )  
;  

```

A.2 Parser

```
unit dpgDpgParser;  
  
parser TdpgDpgParser;  
options  
{  
    defaultErrorHandler = false;  
    importVocab          = dpgDpgLexer;  
    k                    = 2;  
}  
  
// -----  
// grammar  
// -----  
grammar  
    : "unit" id SEMI  
      (usesDecl)?  
      (constDecl)?  
      (typeDecl)?  
      classDecl  
    ;  
  
// -----  
// usesDecl  
// -----  
usesDecl  
    : USES  
      (  
          TOKENREF SEMI  
        | RULEREFS SEMI  
      ) *  
      RCURLY  
    ;  
  
// -----  
// constDecl  
// -----  
constDecl  
    : "const" ACTION  
    ;  
  
// -----  
// typeDecl  
// -----  
typeDecl  
    : "type" ACTION  
    ;
```

```
// -----  
// classDecl  
// -----  
classDecl  
local  
{  
    grType: integer;  
}  
:  
    // -----  
    // Determine parser type  
    // -----  
    (  
        "lexer"      { grType := 0; }  
    |  
        "parser"     { grType := 1; }  
    )  
  
    // -----  
    // get class name  
    // -----  
    id  
    SEMI  
  
    // -----  
    // Process optional class "options {...}" clause  
    // -----  
    (classOptions)?  
  
    // -----  
    // Process optional class "tokens {...}" clause  
    // But only for lexers.  
    // -----  
    ( {grType=0}? classTokens)?  
  
    // -----  
    // Process optional class "memberDecl {...}" clause  
    // -----  
    (classMemberDecl)?  
  
    // -----  
    // Well, the rules  
    // -----  
    rules  
  
    // -----  
    // Process optional class "memberDecl {...}" clause  
    // -----  
    (classMemberDef)?  
;
```

```
// -----  
// classOptions  
// -----  
classOptions  
    :  OPTIONS ( id ASSIGN optionValue SEMI )* RCURLY  
    ;  
  
// -----  
// classTokens  
// -----  
classTokens  
    :  
        TOKENS  
        (  
            TOKENREF SEMI  
            | STRINGLIT SEMI  
        ) *  
  
        RCURLY  
    ;  
  
// -----  
// classMemberDecl  
// -----  
classMemberDecl  
    :  "memberDecl" ACTION  
    ;  
  
// -----  
// classMemberDef  
// -----  
classMemberDef  
    :  "memberDef" ACTION  
    ;  
  
// -----  
// rules  
// -----  
rules  
    :  (rule) *  
    ;  
  
// -----  
// ruleExceptionBlock  
// -----  
ruleExceptionBlock  
    :  "except" ACTION  
    |  "finally" ACTION  
    ;
```

```
// -----  
// altExceptionBlock  
// -----  
altExceptionBlock  
  : "except"    ACTION  
  | "finally"  ACTION  
  ;  
  
// -----  
// rule  
// -----  
rule  
  :  
    // -----  
    // Parse rule scope  
    // -----  
    ( "public"  
      | "protected"  
      | "private"  
    )?  
  
    // -----  
    // Parse rule name  
    // -----  
    id  
  
    // -----  
    // Optional arguments  
    // -----  
    (ARGACTION)?  
  
    // -----  
    // Optional return type  
    // -----  
    ("returns" ARGACTION)?  
  
    // -----  
    // Optional rule options  
    // -----  
    (ruleOptions)?  
  
    // -----  
    // Optional rule local variable declarations  
    // -----  
    ("local" ACTION)?  
  
    // -----  
    // Optional rule init action  
    // -----
```

```

        (ACTION)?

        // -----
        // Rule block
        // -----
        COLON
        block
        SEMI

        // -----
        // Optional exception handler
        // -----
        (ruleExceptionBlock)?
    ;

// -----
// block
// -----
block
    : alternative (OR alternative)*
    ;

// -----
// alternative
// -----
alternative
    : (elem)*
      (altExceptionBlock)?
    ;

// -----
// elem
// -----
elem
    : element
    ;

// -----
// element
// -----
element
local
{
    assignLabel : IdpgToken;
}
{
    assignLabel := nil;
}
    :

```



```
(
    id ASSIGN
    (id COLON)?
    (
        RULEREf (ARGACTION)? (BANG)?
        | TOKENREF (ARGACTION)?
    )
)
|
(assignLabel=id COLON)?
(
    RULEREf (ARGACTION)? (BANG)?
    | range[assignLabel]
    | terminal[assignLabel]
    | NOT (notTerminal[assignLabel] | ebnf[ assignLabel, true])
    | ebnf[ assignLabel, false]
)
| ACTION
| SEMPRED
;

// -----
// range
// -----
range [pTokenLabel: IdpgToken]
local
:
    CHARLIT RANGE CHARLIT
    | (TOKENREF | STRINGLIT) RANGE (TOKENREF | STRINGLIT)
;

// -----
// terminal
// -----
terminal [pTokenLabel: IdpgToken]
:
    CHARLIT (BANG)?
    | TOKENREF (BANG)? (ARGACTION)?
    | STRINGLIT (BANG)?
    | WILDCARD (BANG)?
;

// -----
// notTerminal
// -----
notTerminal [pTokenLabel: IdpgToken]
:
    CHARLIT (BANG)?
    | TOKENREF (BANG)?
;
;
```

```
// -----
// ebnf
// -----
ebnf [pTokenLabel: IdpgToken; pTokenNot: boolean]
  : LPAREN
    (
      subRuleOptions (ACTION)? COLON
    | ACTION COLON
    )?

    block
  RPAREN
  ( QUEST
  | STAR
  | PLUS
  | IMPLIES
  )?
  ;

// -----
// subruleOptions
// -----
subruleOptions
  : OPTIONS (id ASSIGN optionValue)* SEMI RCURLY
  ;

// -----
// ruleOptions
// -----
ruleOptions
  : OPTIONS (id ASSIGN optionValue)* SEMI RCURLY
  ;

// -----
// optionValue
// -----
optionValue returns [IdpgToken]
  : result=qualifiedId
  | result:STRINGLIT
  | result:CHARLIT
  | result:INTEGER
  ;

// -----
// qualifiedId
// -----
qualifiedId returns [IdpgToken]
  : id (WILDCARD id)*
  ;
```

```
// -----  
// id  
// -----  
id returns [IdpgToken]  
  : result:TOKENREF  
  | result:RULEREf  
  ;
```