

# APaGeD v0.4 Documentation

Jascha Wetzel

September 28, 2007

---

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Setting up</b>	<b>4</b>
<b>3</b>	<b>APaGeD grammars</b>	<b>4</b>
<b>4</b>	<b>The Abstract Syntax Tree</b>	<b>7</b>
<b>5</b>	<b>Semantic code</b>	<b>8</b>
<b>6</b>	<b>Initiating semantic analysis</b>	<b>11</b>
<b>7</b>	<b>The Prolog</b>	<b>11</b>
7.1	Global D code blocks . . . . .	11
7.2	Lexeme blocks . . . . .	12
7.3	Imports . . . . .	13
<b>8</b>	<b>Whitespace</b>	<b>14</b>
<b>9</b>	<b>Regular expressions</b>	<b>15</b>
<b>10</b>	<b>Handling conflicts</b>	<b>16</b>
<b>11</b>	<b>Error handling</b>	<b>17</b>
11.1	Detailed LR error messages . . . . .	17
11.2	Synchronization and error recovery . . . . .	18
<b>12</b>	<b>Debugging</b>	<b>19</b>
12.1	Debugging the LL parser . . . . .	20
<b>13</b>	<b>Exporting XML grammar specification</b>	<b>21</b>

# 1 Introduction

APaGeD stands for *Attributed Parser Generator for D*. It let's you write easy-to-read grammars and integrate semantic code for syntax tree traversal right with the syntactical rules.

Although it can generate GLR as well as LL parsers, the emphasis is now on the GLR parsers. This is mainly because they allow more flexible grammars, they have better performance and they provide good error handling with little effort. On the other hand, APaGeD does a lot to tackle the most stated disadvantage of LR parsers, which is the difficulty of debugging them.

This has come to a point where I feel, that debugging the LR parsers can even be easier than debugging an LL parser, because there is more, but not too much information available to analyze the situation. Of course, you need to understand how LR parsers work in order to be able to effectively debug them. And LR parsers are more difficult to understand.

Here is an overview of the features:

- Generates LL and LR parsers

The LL parsers are backtracking top-down parsers with arbitrary lookahead. The LR parsers are generic LR (GLR) parsers based on LALR(1) tables.

- Supports non-regular whitespace

For convenience, you can define whitespace separately from your grammar, such that you don't have to mention whitespace all over your rules. Many parser generators let you define whitespace only as a regular expression, though, effectively prohibiting nested comments, for example. APaGeD uses a secondary grammar to define whitespace, giving you maximum flexibility.

- Includes a linear-time lexical analyzer based on regular expressions using tagged DFAs

Most lexical analyzers use backtracking to find the first longest match, which has a worst-case runtime that is exponential in the length of the input.

- Full multi-pass semantical analysis

Most parser generators allow you to write actions that are executed whenever a non-terminal symbol is expanded (or reduced). Some let you return values from such an action, effectively enabling semantic analysis of s-attributed grammars (only synthetic attributes).

APaGeD gives you full control of the semantic analysis phase, letting you evaluate all non-cyclic attributed grammars. It does so by letting you choose the order in which the non-terminals on the right-hand side of a rule get evaluated on a per-rule basis.

- Syntax is strongly oriented at curly-brace languages and makes grammar files look like an intuitive extension to plain D code

- Imports for grammar files
- Error recovery
- Automatically generated error messages

Unlike many other parser generators that allow semantic actions, APaGeD parses the input completely before applying any semantic code. That is a limitation, because you cannot change the parse tree depending on semantics. APaGeD follows D's philosophy of a strict separation of syntax and semantics here. It is generally favorable to have that separation, as it makes dealing with the grammar significantly easier and faster.

## 2 Setting up

The APaGeD distribution contains

- the APaGeD source code
- the TDFA regular expression engine source code
- build scripts for Windows and Unix
- examples
- this documentation

To build APaGeD, you need D 2.0. Use the included build script. The parsers APaGeD generates can be built with D1.0 or D2.0.

After that, you can compile .apd files like this:

```
apaged grammar.apd parser.d
```

Run APaGeD without any arguments to get a list of command line options. All options are used for debugging only. See the section on debugging for details.

## 3 APaGeD grammars

An APaGeD grammar file consists of rule sets that basically look like function declarations in D, except that they don't have a return type:

```
NonTerminal(out char[] someParameters)
{
    <rule alternatives...>
}
```

Each such block contains all rules for a given non-terminal symbol of your grammar. Each non-terminal has a set of parameters that make up the inherited and synthetic attributes for the semantic phase. We'll see that in a while in more detail.

Each rule is a sequence of non-terminal, terminal symbols or special keywords that represent groups of terminals, just like in a BNF-like grammar description. The non-terminal symbols have to be declared somewhere as shown above. Terminal symbols are either strings in double-quotes or regular expressions.

```
A()
{
    "qwer" "asdf";
    "1234" "asdf";
}
```

To separate rules, the end of a rule has to be marked with a semicolon. The example above therefore specifies two alternative rules with two terminal symbols each.

The keyword `epsilon` specifies, that the non-terminal may also match the empty word, meaning that it doesn't have to match anything at all to succeed. `epsilon` is only allowed as the only symbol in a rule. That is, it is illegal to have a rule like this:

```
A()
{
    epsilon "asdf";
}
```

Furthermore, only one `epsilon`-Rule is allowed per non-terminal and it has to be the last rule.

Regular expressions are marked by wrapping a string in brackets and preceding it with the keyword `regexp` like this:

```
regexp (" [a-zA-Z] [a-zA-Z0-9_]* ")
```

Strings in single quotes are WYSIWYG strings. In a WYSIWYG string, only the single quote itself has to be escaped with a backslash:

```
regexp (' "[^"\n\r]* "' )
```

To simplify access to a regular expression's match, like `epsilon` rules, they are only allowed as the only symbol in a rule. If you want a regular expression to be part of a larger sequence of symbols, simply wrap the regular expression in another non-terminal and use that one instead:

```
A()
{
    NT1 B "asdf" NT2;
}

B()
{
    regexp (" [a-zA-Z] [a-zA-Z0-9_]* " );
}
```

Terminal classes can be defined with the `[]` operator, similar to character classes in regular expressions. For example,

```
[ "{ " } " ]
```

matches one of the two terminals.

```
[ ^ "{ " } " ]
```

matches any of the terminals that appear in the grammar, except the two curly braces. Like `regex`, the `[]` operator has to be the only symbol in a rule.

Symbol sequences can certainly be distributed over multiple lines:

```
A()
{
    NT1 B
    "asdf" NT2;
}
```

APaGeD allows accessing the parser's lookahead in the grammar. Matching a non-terminal can be restricted to a set of first terminals. That is, the non-terminal will only be matched, if the first terminal that it expands to is one of those specified in a terminal class preceded by `>`:

```
A()
{
    "asdf" More;
    "qwer" More;
}

B()
{
    >["asdf"] A;
}
```

B will only match the A's first rule.

Here is a simple example for a grammar that specifies arithmetic expression on integers that consider operator precedence (`*` binds stronger than `+` and `-`).

```
Expr()
{
    MulExpr "+" Expr;
    MulExpr "-" Expr;
    MulExpr;
}

MulExpr()
{
```

```

    Atom "*" MulExpr;
    Atom;
}

Atom()
{
    regexp("-?[0-9]+");
    "(" Expr ";";
}

```

You can already compile that grammar with APaGeD. The first non-terminal in a grammar file will be treated as the start symbol for the grammar. APaGeD will create a D source file that contains a global function

```

bool parse(
    string          filename,
    ref string      input,
    out SyntaxTree* root,
    bool            detailed = false,
    bool            recover  = false
)

```

- `filename` is only used in error messages and can therefore also be null.
- `input` obviously is the input that will be parsed.
- `root` is a pointer to an instance of the `SyntaxTree` structure that will receive the root node of the resulting abstract syntax tree.
- `detailed` toggles whether error messages will contain detailed status output or just the filename, line- and column number and error message.
- `recover` enables error recovery.

`parse` returns `true` if parsing was successful. That means that the grammar matched *all* of the input.

## 4 The Abstract Syntax Tree

The `SyntaxTree` structure, also defined in the generated D source file, basically looks like this:

```

struct SyntaxTree
{
    uint    _ST_rule,
           _ST_line_number,
           _ST_column_number;
}

```

```

union
{
    struct {
        string  _ST_match,
               _ST_match_ws;
    }
    SyntaxTree*[] _ST_children;
}

// functions
}

```

An AST node for a non-terminal contains pointers to its children. A leaf node contains the terminal match of the rule that generated the node<sup>1</sup>.

`_ST_match_ws` is the match including preceding whitespace.

`_ST_rule` is the number of the rule. The numbering of the rules can be seen in the grammar debug output (command line option `-g`).

`_ST_line_number` and `_ST_column_number` are the line- and column numbers of the first character of the match. If the node is a non-leaf node, it's the line- and column number of the first terminal in the subtree rooted at that node, that is, the first terminal of the given rule's expansion.

The tree is used to do semantical analysis. The only thing you need to know for common use, is that you can use these member variables in your semantic code. Usually that will only be `_ST_match` and `_ST_line_number`.

The structure also defines all the semantic code. APaGeD extends the blocks of code that you write after each rule such that you don't have to care about which code to call for which rule, etc.

## 5 Semantic code

Instead of terminating a rule with a semicolon, you can append a block of D code:

```

A()
{
    "asdf"
    {
        // D code
    }
}

```

Within these code blocks, you can use the member variables of the `SyntaxTree` structure, as well as the names of the non-terminals used in the rule. These non-terminal names are delegates that point to the corresponding code block of the rule that was reduced to the respective instance of the non-terminal. The use is very straight forward, as you will see in a moment.

---

<sup>1</sup>note that non-terminal leaf nodes don't make any sense, but they would contain a match, as well

Let's expand the arithmetic example from above to actually calculate the value of an expression. We will do so by using a single synthetic attribute `value` for all non-terminals. That means that we declare an output parameter in each non-terminal that will be computed bottom-up when traversing the tree.

The simplest case therefore is a number. We know the value of those immediately:

```
Atom(out int value)
{
    regexp("-?[0-9]+")
    {
        value = atoi(_ST_regexMatch);
    }
}
```

Here we use the `_ST_match` member to access the string that has been matched by the regular expression.

To use `atoi` we would have to import `std.string` first. In this case, the generated D source file already imports it, so we don't have to do it manually. But we'll see how to add our own imports and global declarations in a later section.

Now that `Atom` can give use values, we can use them in non-leaf nodes. As mentioned above, we can call non-terminal symbols just as if they were normal functions. Therefore, the semantic code is straight forward:

```
MulExpr(out int value)
{
    Atom "*" MulExpr
    {
        MulExpr(value);
        int tmp;
        Atom(tmp);
        value *= tmp;
    }
}
```

First we call `MulExpr` to compute the value of the right side of the multiplication and save the value the the output parameter. Then we call `Atom` with a temporary variable and multiply both values to produce the result. Note that we don't care about traversing the syntax tree. `APaGeD` takes care of this for us.

The code for the `Expr` non-terminal looks similar:

```
Expr(out int value)
{
    MulExpr "+" Expr
    {
        int val;
        Expr(val);
        MulExpr(value);
    }
}
```

```

        value += val;
    }

MulExpr "-" Expr
{
    int val;
    Expr(val);
    MulExpr(value);
    value -= val;
}
}

```

See the `arithmetic.apd` example for the full version.

If a non-terminal symbol appears more than once in a rule, you need to specify aliases for all instances but one to make the semantic calls unique. This is done by appending `=<alias>` to the non-terminal:

```

MulExpr "*" MulExpr=MulExpr2
{
    MulExpr(value); // calls the first instance
    int tmp;
    MulExpr2(tmp); // calls the second instance
    value *= tmp;
}

```

Now it is obvious how we're going to use inherited attributes, that means values that are computed top-down. We'll simply add parameters to our non-terminals without the `out` modifier. Those will be available in semantic code blocks just like normal function parameters. Of course you can also use `inout` parameters or any other valid D parameter declaration.

You can also call non-terminals multiple times within the same semantic code block, effectively performing multi-pass semantic analysis. APaGeD does this for example to implement forward references for non-terminals.

See the `binfloat.apd` example and `src/parser.apd` for details.

The declaration of a non-terminal may be followed by attributes. At the moment, the only attribute available is `no_ast`, which prevents AST nodes to be generated for that non-terminal and all of its children.

```

A() no_ast
{
    "asdf";
}

```

Note that semantic code will not be executed, if a non-terminal has no AST node. Not generating AST nodes improves parsing speed and simplifies the AST.

## 6 Initiating semantic analysis

After successfully calling the `parse` function, you'll have the root node of the `SyntaxTree`. It will contain a function with the same name as your start symbol (the first non-terminal defined in the grammar file). Parsing and initiating semantic analysis for our arithmetic example therefore looks like this:

```
SyntaxTree* root;
if ( parse(filename, input, root) ) {
    int value;
    root.Expr(value);
    writefln("Result: %d", value);
}
else
    writefln("Invalid expression: %s", input);
```

## 7 The Prolog

As mentioned before, you can add global code blocks to an APaGeD grammar. You can import external grammar files and have explicit blocks that define lexemes. As of version 0.3, all of these declarations have to appear before the non-terminals in a grammar file and are therefore called the prolog. Within the prolog, multiple instances of the prolog declarations (Declaration blocks, imports, lexemes or properties) may appear.

The only property available at the moment is `parser_type`. By default it's value is `lr` which makes APaGeD generate GLR parsers. Setting it to `ll` will cause generation of LL parsers:

```
APDProperties {
    parser_type = ll
}
```

### 7.1 Global D code blocks

Declaration blocks may appear in a non-terminal body and in the prolog. The declaration blocks in the prolog will be copied to the generated code before any semantic rule code. Declaration blocks start with the keyword `APDDeclaration` followed by a D code block:

```
APDDeclaration
{
    import other.stuff;

    void main()
    {
        // do something useful
    }
}
```

```

    }
}

```

Within the body of a non-terminal, a declaration block may appear once, before any rule:

```

BinFloat(out real value)
{
    APDDeclaration {
        uint p = 0, val;
    }

    BitList {
        BitList(p, val);
        value = val;
    }
}

```

Everything that is placed in a Declaration block at the beginning of a non-terminal, will be declared and executed for all of the rules.

## 7.2 Lexeme blocks

The order of appearance of the grammar's lexemes is used as the order for the first-longest-match decision. In a programming language, keyword lexemes have therefore to appear before an identifier lexeme. Else, the lexical analyzer will always match the identifier and never a keyword.

This may dictate the order of the non-terminals, which may not be desirable. To work around this, some lexemes can optionally be specified in a lexeme block. The lexemes in the lexeme block will be listed before any implicitly collected lexeme from the non-terminal declarations. The order in the lexeme blocks is certainly maintained.

A lexeme block starts with the keyword `APDLexemes` followed by a block separated by curly braces containing a list of lexemes.

```

APDLexemes
{
    regexp("[,\\.\\*\\/\\+\\-\\%$~\\|&~'\\\\\\\\\"0-9<\\>\\?:]")
    "asdf"
}

```

Additionally, lexemes that do not appear anywhere else in the grammar may be specified in a lexeme block. This may be useful in combination with the `[]` operator<sup>2</sup>.

Sometimes a context free grammar is not enough to parse the lexical structure. An example are HEREDOC strings that allow custom delimiters. That is, `MyDelim...MyDelim`

---

<sup>2</sup>APaGeD makes use of that, because the lexer has to match all D lexemes, while the APaGeD grammar itself only uses a small subset

represents the string . . . and `MyDelim` can be chosen freely. Such a delimited string is a context sensitive construct. To parse it we need an attributed grammar. Using an attributed grammar for lexical analysis would be a waste of performance, though. Especially, since usually, there aren't many lexemes that require context sensitive parsing.

```
APDLexemes
{
    'q"'
    {
        // lexer code
    }
}
```

The custom lexer code is inserted into the parser function. Consult the generated source code and the custom lexer code in the SEATD grammar for details.

### 7.3 Imports

An import declaration starts with the keyword `import` followed by a module name and terminated with a semicolon:

```
import basics.lexical;
```

The module name is a relative file path with dots used as separators, rather than front- or backslashes. These file paths have to be relative to the include path, which can be specified with the `-I` command line option. The current directory is always part of the include path.

The order in which the imports appear affects the order of the Declaration blocks in the generated code as well as the order of the lexemes. The latter has to be dealt with carefully, since it may change the behaviour of the parser. The order of the imported declarations and lexeme blocks is the same as if the import declaration was substituted with the imported module, but prolog and non-terminal declarations are separated. That means, that all lexemes from lexeme blocks will still appear before any implicitly collected lexeme.

The following is also possible:

```
APDLexemes {
    "asdf"
}

import other;

APDLexemes {
    "qwer"
}
```

Assuming, that the module `other` has a lexeme block, too, the order of the lexemes will be:

- "asdf"
- lexemes from lexeme block in module `other`
- "qwer"
- lexemes from non-terminals in module `other`
- lexemes from non-terminals in main module

## 8 Whitespace

APaGeD can automatically remove whitespace from the input before reading the next lexeme from the input, such that you don't have to specify whitespace symbols all over your grammar. You can still do that, if your grammar does not allow whitespace to appear in arbitrary places<sup>3</sup>.

To define what whitespace is, a secondary grammar is used. Since whitespace does not need semantics<sup>4</sup>, the specification of the whitespace grammar is separated from the main grammar by simply omitting the parameters and semantic code blocks in a non-terminal declaration. A non-terminal of the whitespace grammar therefore looks like this:

```
Whitespace
{
    regexp("[\\n\\r\\t ]+");
}
```

The above example is already enough to match simple whitespace. Obviously, in this case, a single regular expression would be enough. For more complex whitespace grammars with comments, we need a context-free grammar, though. Here is an example that matches D style whitespace, including nested comments:

```
Whitespace
{
    Whitespace WhitespaceFlat;
    WhitespaceFlat;
}

WhitespaceFlat
{
    regexp("[\\n\\r\\t ]+");
    regexp("//[^\n]*");
    regexp("/\\*([^\n]|\\*>/)*\\*/");
    "/"+" WhitespaceNesteds "+"/";
}
```

<sup>3</sup>like XML does, for example

<sup>4</sup>if you want to parse documentation comments, put them into the main grammar

```

WhitespaceNesteds
{
    WhitespaceNesteds WhitespaceNested;
    WhitespaceNested;
}

WhitespaceNested
{
    WhitespaceFlat;
    regexp("[^#/\\+\\*\\n\\r\\t ]+");
    "+";
    "*";
    "/";
}

```

Note, that the first `regexp` in `WhitespaceFlat` uses the special lookahead operator `>`. See the section on regular expressions for details.

The separation of the lexemes in `WhitespaceNested` is necessary due to the first-longest-match behaviour of the lexical analyzer.

## 9 Regular expressions

APaGeD includes a regular expression compiler that generates linear-time matchers<sup>5</sup>.

Here is a short overview of the supported operators:

- `|` alternation
- `(...)` non-matching brackets
- `(?...)` matching brackets (usually not used in lexemes)
- `?` zero or one repetition (greedy)
- `*` zero or more repetitions (greedy)
- `+` one or more repetitions (greedy)
- `??` zero or one repetition (reluctant)
- `*?` zero or more repetitions (reluctant)
- `+?` one or more repetitions (reluctant)
- `{x, y}` counted occurrence (greedy). At least  $x$ , at most  $y$  occurrences. Omitting  $x$  is equivalent to  $x = 0$ , omitting  $y$  is equivalent to  $y = \infty$ .

---

<sup>5</sup>most regular expression engines use backtracking, which generally has exponential runtime complexity

- `{x, y}?` counted occurrence (reluctant)
- `.` any character (precisely: 0x09-0x13, 0x20-0x7e, 0xa0-0xff, 0x0100-0x017f, 0x0180-0x024f, 0x20a3-0x20b5)
- `[...]` `[^...]` character classes
- `>x` negative, single character lookahead (character classes not supported for *x*, yet)

The lookahead is a speciality of the APaGeD implementation. It is a lot faster than general lookahead<sup>6</sup> but less powerful. For many situations it is powerful enough, though. For example, `\*>/` matches any `*` that is not followed by a `/`.

The syntax for matching and non-matching brackets has been switched for convenience, since sub-matches are usually not used in lexemes.

## 10 Handling conflicts

Although APaGeD generates generic LR parsers, that can parse any context free grammar regardless of the conflicts, that the underlying LR tables may have, it may be useful to resolve conflicts to improve the parser's performance.

APaGeD lists details for all conflicts when given the `-c` command line option.

Conflict resolution can be controlled with three precedence attributes `force`, `deny` and `prefer`. These attributes can follow the header of a non-terminal or the symbols of a rule. If applied to the non-terminal, all rules will use the specified precedence attributes. The precedence attributes specify the parser's behaviour when it encounters a conflict that involves the non-terminal or rule for which the precedence attribute has been specified.

`force` makes the parser ignore the other branches in a conflict, making this non-terminal or rule the only choice.

`deny` makes the parser ignore the branch for this non-terminal or rule and only consider the remaining ones.

`prefer` makes the parser prefer this branch over the others. It will still try all of them, but the order changes. This is useful since many conflicts have branches that are statistically more likely to be correct than others.

Each of the precedence attributes can be followed by a list of symbols. The precedence rules will only consider those branches of the conflict, that

- in a reduce/reduce conflict, correspond to rules that belong to non-terminals that are listed, or
- in a shift/reduce conflict, are the terminal symbol that may be shifted.

To utilize the `prefer` attribute, it is useful to measure the performance of conflicts and create the statistic of their branches' success. When the parser is compiled with

---

<sup>6</sup>like the one Perl regular expressions uses, for example

`-version=ProfileConflicts`, the parser will count how often a conflicts was encountered and how often on of the branches failed. In `examples/profile_conflicts.d` is a code snippet, that will extract that data frmo a `GLRParser` object and print the results.

## 11 Error handling

APaGeD automatically generates error messages of the form

```
filename(line number): found A, expected B, C or D
```

If you want to specify your own error messages, you can add a special operator in your rules. It is similar to PROLOG's cut operator (if you happen to know it):

```
Program()
{
    "begin" !("Statement expected after begin in line %d")
    Statement "end"
    {}
}
```

The effect of this operator is that if a `Program` symbol is being parsed and the `"begin"` terminal has been matched successfully, but the `Statement` non-terminal fails, parsing is aborted and a `ParserException` with the given error message is thrown.

The `! ("...")` operator therefore cuts off the search tree at it's position. If the parser is in a non-deterministic state, that is, there might be different ways to parse the input, it will not backtrack to try the other paths. If no explicit error message is given, it will backtrack.

### 11.1 Detailed LR error messages

As described before, the `parse` function can be instructed to create detailed error messages. They look like this:

```
Error: src\parser.apd(27): found ";", expected DCodeBlock
input: ;\n{\n    class LexemeSet\n    {\n        Set!(
lookahead: ;
lexeme: ;
AST node stack:
Prolog
LR stack:
---- State 2 (1) ----
Start -> Prolog . Grammar
Prolog -> Prolog . import FQIdent ;
Prolog -> Prolog . APDProperties \{ Props \}
```

```

Prolog -> Prolog . APDLexemes \{ Lexs \}
Prolog -> Prolog . APDDeclaration DCodeBlock
---- State 39 (27) ----
Prolog -> Prolog APDDeclaration . DCodeBlock

```

The first line is the standard error message. The remaining lines are detail.

`input` prints the portion of the input, that the parser is currently looking at.

`lookahead` shows current lookahead match.

`lexeme` is the lexeme that matched the lookahead.

The following lists the current node stack. In the example only a `Prolog` non-terminal has been reduced, so far.

Finally, the LR stack is pretty printed using the rules the LR states correspond to. An LR parser tries to consider multiple rules at once and narrows down the choice as it reads terminal symbols from the input. A dot in the rule description indicates where, within a rule, the parser is in that state. In the example you can see, that in state 2, the parser has successfully read a `Prolog`. In the next state (state 39), it has successfully read `APDDeclaration`, and is expecting a `DCodeBlock`, but found a semicolon.

In the input you can see, that there is a semicolon in front of the curly brace that opens a `DCodeBlock`. The error is obvious.

The number in parenthesis after the state index is the line number the parser was in, when it was in that state. The line number of the last – the current – state is therefore always the same as in the normal error message.

With this detail information you can very precisely track down parser errors, as you can see the path the parser took to get into the error situation. You can see all alternatives (the different rules in each state) the parser had on it's way there, but no more.

The LR stack will be short, if you use left recursion as much as possible<sup>7</sup>. That helps to make debugging the grammar (or the input) more effective.

## 11.2 Synchronization and error recovery

For an ambiguous grammar, the parser may search multiple paths. If an error occurs in one path, it abandons that path and tries the remaining ones. Very often, the correct path is found before all other paths have been tried, though. If an error occurs on the correct path, the parser will therefore abandon it and try incorrect ones, ultimately leading to false error messages.

To avoid that situation, synchronization points can be set in the grammar. Once passed, the parser will not backtrack beyond that point. Synchronization points are usually put after constructs, that you are certain to be correct, once they have been parsed successfully. The end of a declaration or the end of a statement<sup>8</sup> in a programming language are examples for such points.

To insert a synchronization point in a rule, add the `!(sync)` operator:

```

Decl
{

```

<sup>7</sup>which LR parsers deal with more efficiently for the same reason

<sup>8</sup>after the semicolon, for example

```

    Decls !(sync) Decl;
    Decl;
}

```

Lists of declarations or statements provide typical synchronization points, as shown in the example.

For grammars with conflicts<sup>9</sup>, good placement of synchronization points may be required to obtain useful error messages. If the grammar has no conflicts, synchronization points do not affect the error messages, because there will always be a single path.

Synchronization points are also used for error recovery. APaGeD may recover from errors and try to parse the rest of the file. In order to do so, it ignores the terminal, that produces the error, throws away non-terminal symbols that have been reduced until it reaches a synchronization point and continues parsing.

## 12 Debugging

If you compile the generated D source file with

```
dmd -debug -debug=parser parser.d
```

it will output verbose progress information during parsing. For an LR parser, that output is similar to the detailed error information. See the section on error handling for details.

Adding `-debug=lexer` will additionally print the portion of the input that has been matched by the whitespace grammar in each step.

`-debug=nonfatal` will print error messages of non-fatal errors. Non-fatal are errors if they occur in an ambiguous situation, where the parser has multiple paths that can lead to a correct parse tree.

Besides the statistics printed during generation of the parser, APaGeD can generate additional information about your grammar and the parser that is useful when debugging a grammar.

The `-g` command line option prompts APaGeD to output a compact, numbered listing of all lexemes and production rules, which is also useful as an export for documentation.

`-a` prints the LALR(1) automaton – all states, all LALR(1) entries, all transitions – this can get pretty lengthy. But if in doubt, it's the definitive reference for the parser's behaviour.

`-t` prints the LALR(1) action and goto tables formatted in HTML. Note that some browsers may have problems with very large tables. On the other hand, beyond a certain size, the tables cease to being helpful anyway.

### 12.1 Debugging the LL parser

Compiling an LL parser with `-debug_ -debug=parser`, will output the whole search tree, that looks like this:

---

<sup>9</sup>the statistics that APaGeD prints when compiling the grammar, show whether there were conflicts

```

Expr "4+*2"
  MulExpr "4+*2"
    Atom "4+*2"
    *Atom "+*2"
    Atom "4+*2"
    *Atom "+*2"
  *MulExpr "+*2"
    Expr "*2"
      MulExpr "*2"
        Atom "*2"
        -Atom "*2"
        Atom "*2"
        -Atom "*2"
      -MulExpr "*2"
        MulExpr "*2"
          Atom "*2"
          -Atom "*2"
          Atom "*2"
          -Atom "*2"
        -MulExpr "*2"
          MulExpr "*2"
            Atom "*2"
            -Atom "*2"
            Atom "*2"
            -Atom "*2"
          -MulExpr "*2"
            -Expr "*2"
              MulExpr "4+*2"
                Atom "4+*2"
                *Atom "+*2"
                Atom "4+*2"
                *Atom "+*2"
              *MulExpr "+*2"
                MulExpr "4+*2"
                  Atom "4+*2"
                  *Atom "+*2"
                  Atom "4+*2"
                  *Atom "+*2"
                *MulExpr "+*2"
              *Expr "+*2"

```

This shows you what calls the recursive descent parser actually made while trying to match the input. First `Expr` with input `4+*2` called `MulExpr`, which in turn called `Atom`, which succeeded, which is denoted by another line with the non-terminal name preceded by a `*`. As you can also see, the first symbol (the `4`) was removed from the input. But since the first rule of `MulExpr` expects an `Atom` to be followed by a `*`, the

rule fails and the next rule is tried. That one succeeds, such that the `MulExpr` rules succeeds.

We're still in the first `Expr` rule, which matches the terminal `+` now, followed by a non-terminal `Expr`, which is again printed in the debug output. After entering the `MulExpr` again, the calls to `Atom` fail this time, which is denoted by a `-` before the second line for the non-terminal call. Therefore `MulExpr` fails, and the `Expr` call tries it's next rule. And so on...

In the end, `Expr` matched only the `4`, the remaining input is `++2`. The parse function will therefore return `false`, although `Expr` returns `true`, since it requires all input to have been matched.

## 13 Exporting XML grammar specification

APaGeD can export the grammar specification formatted in XML. Those files can for example be processed with XSLT to generate HTML. An example XSLT file is in `doc/apd2html.xsl`.

To generate the XML file, pass the `-gx_<filename>` option to APaGeD. Optionally, you can pass `-n` to avoid generating the parser itself.