Parse Tools Application (PARSETOOLS)

version 1.4

Typeset in LaTeX from SGML source using the DocBuilder-0.9.8 Document System.

Contents

Parsetools Reference Manual

Short Summaries

- Erlang Module yecc [page 2] – LALR-1 Parser Generator

yecc

The following functions are exported:

- `file(Grammarfile [, Options]) -> YeccRet`
  [page 2] Give information about resolved and unresolved parse action conflicts.
- `format_error(Reason) -> Chars`
  [page 3] Return an English description of a an error tuple.

yecc

Erlang Module

An LALR-1 parser generator for Erlang, similar to `yacc`. Takes a BNF grammar definition as input, and produces Erlang code for a parser.

To understand this text, you also have to look at the `yacc` documentation in the UNIX(TM) manual. This is most probably necessary in order to understand the idea of a parser generator, and the principle and problems of LALR parsing with finite look-ahead.

Exports

`file(Grammarfile [, Options]) -> YeccRet`

Types:

- Grammarfile = filename()
- Options = Option | [Option]
- Option =-see below-
- YeccRet = {ok, Parserfile} | {ok, Parserfile, Warnings} | error | {error, Warnings, Errors}
- Parserfile = filename()
- Warnings = Errors = [{filename(), [ErrorInfo]}]
- ErrorInfo = {ErrorLine, module(), Reason}
- ErrorLine = integer()
- Reason =-formatable by format_error/1-

`Grammarfile` is the file of declarations and grammar rules. Returns `ok` upon success, or `error` if there are errors. An Erlang file containing the parser is created if there are no errors. The options are:

{`parserfile, Parserfile`}. `Parserfile` is the name of the file that will contain the Erlang parser code that is generated. The default (`""`) is to add the extension `.erl` to `Grammarfile` stripped of the `.yrl` extension.

{`includefile, Includefile`}. Indicates a customized prologue file which the user may want to use instead of the default file `lib/parsetools/include/yeccpre.hrl` which is otherwise included at the beginning of the resulting parser file. *N.B.* The `Includefile` is included 'as is' in the parser file, so it must not have a module declaration of its own, and it should not be compiled. It must, however, contain the necessary export declarations. The default is indicated by `""`.

{`report_errors, bool()`}. Causes errors to be printed as they occur. Default is `true`.

{report_warnings, bool()}. Causes warnings to be printed as they occur. Default is true.

{report, bool()}. This is a short form for both report_errors and report_warnings.

{return_errors, bool()}. If this flag is set, {error, Errors, Warnings} is returned when there are errors. Default is false.

{return_warnings, bool()}. If this flag is set, an extra field containing Warnings is added to the tuple returned upon success. Default is false.

{return, bool()}. This is a short form for both return_errors and return_warnings.

{verbose, bool()}. Determines whether the parser generator should give full information about resolved and unresolved parse action conflicts (true), or only about those conflicts that prevent a parser from being generated from the input grammar (false, the default).

Any of the Boolean options can be set to true by stating the name of the option. For example, verbose is equivalent to {verbose, true}.

The value of the Parserfile option stripped of the .erl extension is used by Yecc as the module name of the generated parser file.

Yecc will add the extension .yrl to the Grammarfile name, the extension .hrl to the Includefile name, and the extension .erl to the Parserfile name, unless the extension is already there.

format_error(Reason) -> Chars

Types:

- Reason =-as returned by yecc:file/1,2-
- Chars = [char() | Chars]

Returns a descriptive string in English of an error tuple returned by yecc:file/1,2. This function is mainly used by the compiler invoking Yecc.

Pre-Processing

A scanner to pre-process the text (program, etc.) to be parsed is not provided in the yecc module. The scanner serves as a kind of lexicon look-up routine. It is possible to write a grammar that uses only character tokens as terminal symbols, thereby eliminating the need for a scanner, but this would make the parser larger and slower.

The user should implement a scanner that segments the input text, and turns it into one or more lists of tokens. Each token should be a tuple containing information about syntactic category, position in the text (e.g. line number), and the actual terminal symbol found in the text: {Category, LineNumber, Symbol}.

If a terminal symbol is the only member of a category, and the symbol name is identical to the category name, the token format may be {Symbol, LineNumber}.

A list of tokens produced by the scanner should end with a special end_of_input tuple which the parser is looking for. The format of this tuple should be {Endsymbol, LastLineNumber}, where Endsymbol is an identifier that is distinguished from all the terminal and non-terminal categories of the syntax rules. The Endsymbol may be declared in the grammar file (see below).

The simplest case is to segment the input string into a list of identifiers (atoms) and use those atoms both as categories and values of the tokens. For example, the input string `aaa bbb 777, X` may be scanned (tokenized) as:

```
[{aaa, 1}, {bbb, 1}, {777, 1}, {',' , 1}, {'X', 1},
 {'$end', 1}].
```

This assumes that this is the first line of the input text, and that `'$end'` is the distinguished `end_of_input` symbol.

The Erlang scanner in the `io` module can be used as a starting point when writing a new scanner. Study `yeccscan.erl` in order to see how a filter can be added on top of `io:scan_erl_form/3` to provide a scanner for Yecc that tokenizes grammar files before parsing them with the Yecc parser. A more general approach to scanner implementation is to use a scanner generator. A scanner generator in Erlang called `leex` is under development.

Grammar Definition Format

Erlang style `comments`, starting with a `'%'`, are allowed in grammar files.

Each `declaration` or `rule` ends with a dot (the character `'.'`).

The grammar starts with a declaration of the `nonterminal categories` to be used in the rules. For example:

```
Nonterminals sentence nounphrase verbphrase.
```

A non-terminal category can be used at the left hand side (= `lhs`, or `head`) of a grammar rule. It can also appear at the right hand side of rules.

Next comes a declaration of the `terminal categories`, which are the categories of tokens produced by the scanner. For example:

```
Terminals article adjective noun verb.
```

Terminal categories may only appear in the right hand sides (= `rhs`) of grammar rules.

Next comes a declaration of the `rootsymbol`, or start category of the grammar. For example:

```
Rootsymbol sentence.
```

This symbol should appear in the lhs of at least one grammar rule. This is the most general syntactic category which the parser ultimately will parse every input string into.

After the rootsymbol declaration comes an optional declaration of the `end_of_input` symbol that your scanner is expected to use. For example:

```
Endsymbol '$end'.
```

Next comes one or more declarations of `operator precedences`, if needed. These are used to resolve shift/reduce conflicts (see `yacc` documentation).

Examples of operator declarations:

```
Right 100 '='.
Nonassoc 200 '==' '=/='.
Left 300 '+'.
Left 400 '*'.
Unary 500 '-'.
```

These declarations mean that '=' is defined as a `right associative binary` operator with precedence 100, '==' and '=/=' are operators with `no associativity`, '+' and '*' are `left associative binary` operators, where '*' takes precedence over '+' (the normal case), and '-' is a `unary` operator of higher precedence than '*'. The fact that '==' has no associativity means that an expression like `a == b == c` is considered a syntax error.

Certain rules are assigned precedence: each rule gets its precedence from the last terminal symbol mentioned in the right hand side of the rule. It is also possible to declare precedence for non-terminals, "one level up". This is practical when an operator is overloaded (see also example 3 below).

Next come the `grammar rules`. Each rule has the general form

```
Left_hand_side -> Right_hand_side : Associated_code.
```

The left hand side is a non-terminal category. The right hand side is a sequence of one or more non-terminal or terminal symbols with spaces between. The associated code is a sequence of zero or more Erlang expressions (with commas ',' as separators). If the associated code is empty, the separating colon ':' is also omitted. A final dot marks the end of the rule.

Symbols such as '{', '.', etc., have to be enclosed in single quotes when used as terminal or non-terminal symbols in grammar rules. The use of the symbols '$empty', '$end', and '$undefined' should be avoided.

The last part of the grammar file is an optional section with Erlang code (= function definitions) which is included 'as is' in the resulting parser file. This section must start with the pseudo declaration, or key words

```
Erlang code.
```

No syntax rule definitions or other declarations may follow this section. To avoid conflicts with internal variables, do not use variable names beginning with two underscore characters ('__') in the Erlang code in this section, or in the code associated with the individual syntax rules.

The optional `expect` declaration can be placed anywhere before the last optional section with Erlang code. It is used for suppressing the warning about conflicts that is ordinarily given if the grammar is ambiguous. An example:

```
Expect 2.
```

The warning is given if the number of shift/reduce conflicts differs from 2, or if there are reduce/reduce conflicts.

Examples

A grammar to parse list expressions (with empty associated code):

```
Nonterminals list elements element.
Terminals atom '(' ')'.
Rootsymbol list.
list -> '(' ')'.
list -> '(' elements ')'.
elements -> element.
elements -> element elements.
element -> atom.
element -> list.
```

This grammar can be used to generate a parser which parses list expressions, such as `()`, `(a)`, `(peter charles)`, `(a (b c) d ((()))`, ... provided that your scanner tokenizes, for example, the input `(peter charles)` as follows:

```
[{'(', 1} , {atom, 1, peter}, {atom, 1, charles}, {')', 1},
 {'$end', 1}]
```

When a grammar rule is used by the parser to parse (part of) the input string as a grammatical phrase, the associated code is evaluated, and the value of the last expression becomes the value of the parsed phrase. This value may be used by the parser later to build structures that are values of higher phrases of which the current phrase is a part. The values initially associated with terminal category phrases, i.e. input tokens, are the token tuples themselves.

Below is an example of the grammar above with structure building code added:

```
list -> '(' ')' : nil.
list -> '(' elements ')' : '$2'.
elements -> element : {cons, '$1', nil}.
elements -> element elements : {cons, '$1', '$2'}.
element -> atom : '$1'.
element -> list : '$1'.
```

With this code added to the grammar rules, the parser produces the following value (structure) when parsing the input string `(a b c)`.. This still assumes that this was the first input line that the scanner tokenized:

```
{cons, {atom, 1, a,} {cons, {atom, 1, b},
                            {cons, {atom, 1, c}, nil}}}
```

The associated code contains pseudo variables`'$1'`, `'$2'`, `'$3'`, etc. which refer to (are bound to) the values associated previously by the parser with the symbols of the right hand side of the rule. When these symbols are terminal categories, the values are token tuples of the input string (see above).

The associated code may not only be used to build structures associated with phrases, but may also be used for syntactic and semantic tests, printout actions (for example for tracing), etc. during the parsing process. Since tokens contain positional (line number) information, it is possible to produce error messages which contain line numbers. If there is no associated code after the right hand side of the rule, the value `'$undefined'` is associated with the phrase.

The right hand side of a grammar rule may be empty. This is indicated by using the special symbol `'$empty'` as rhs. Then the list grammar above may be simplified to:

```
list -> '(' elements ')' : '$2'.
elements -> element elements : {cons, '$1', '$2'}.
elements -> '$empty' : nil.
element -> atom : '$1'.
element -> list : '$1'.
```

Generating a Parser

To call the parser generator, use the following command:

```
yecc:file(Grammarfile).
```

An error message from Yecc will be shown if the grammar is not of the LALR type (for example too ambiguous). Shift/reduce conflicts are resolved in favor of shifting if there are no operator precedence declarations. Refer to the yacc documentation on the use of operator precedence.

The output file contains Erlang source code for a parser module with module name equal to the `Parserfile` parameter. After compilation, the parser can be called as follows (the module name is assumed to be `myparser`):

```
myparser:parse(myscanner:scan(Inport))
```

The call format may be different if a customized prologue file has been included when generating the parser instead of the default file `lib/parsetools/include/yeccpre.hrl`.

With the standard prologue, this call will return either `{ok, Result}`, where `Result` is a structure that the Erlang code of the grammar file has built, or `{error, {Line_number, Module, Message}}` if there was a syntax error in the input.

`Message` is something which may be converted into a string by calling `Module:format_error(Message)` and printed with `io:format/3`.

> **Note:**
> By default, the parser that was generated will not print out error messages to the screen. The user will have to do this either by printing the returned error messages, or by inserting tests and print instructions in the Erlang code associated with the syntax rules of the grammar file.

It is also possible to make the parser ask for more input tokens when needed if the following call format is used:

```
myparser:parse_and_scan({Function, Args})
myparser:parse_and_scan({Mod, Tokenizer, Args})
```

The tokenizer `Function` is either a fun or a tuple `{Mod, Tokenizer}`. The call `apply(Function, Args)` or `apply({Mod, Tokenizer}, Args)` is executed whenever a new token is needed. This, for example, makes it possible to parse from a file, token by token.

The tokenizer used above has to be implemented so as to return one of the following:

```
{ok, Tokens, Endline}
{eof, Endline}
{error, Error_description, Endline}
```

This conforms to the format used by the scanner in the Erlang `io` library module.

If `{eof, Endline}` is returned immediately, the call to `parse_and_scan/1` returns `{ok, eof}`. If `{eof, Endline}` is returned before the parser expects end of input, `parse_and_scan/1` will, of course, return an error message (see above). Otherwise `{ok, Result}` is returned.

More Examples

1. A grammar for parsing infix arithmetic expressions into prefix notation, without operator precedence:

```
Nonterminals E T F.
Terminals '+' '*' '(' ')' number.
Rootsymbol E.
E -> E '+' T: ['$1', '$2', '$3'].
E -> T : '$1'.
T -> T '*' F: ['$1', '$2', '$3'].
T -> F : '$1'.
F -> '(' E ')' : '$2'.
F -> number : '$1'.
```

2. The same with operator precedence becomes simpler:

```
Nonterminals E.
Terminals '+' '*' '(' ')' number.
Rootsymbol E.
Left 100 '+'.
Left 200 '*'.
E -> E '+' E : ['$1', '$2', '$3'].
E -> E '*' E : ['$1', '$2', '$3'].
E -> '(' E ')' : '$2'.
E -> number : '$1'.
```

3. An overloaded minus operator:

```
Nonterminals E uminus.
Terminals '*' '-' number.
Rootsymbol E.

Left 100 '-'.
Left 200 '*'.
Unary 300 uminus.

E -> E '-' E.
E -> E '*' E.
E -> uminus.
E -> number.

uminus -> '-' E.
```

4. The Yecc grammar that is used for parsing grammar files, including itself:

```
Nonterminals
grammar declaration rule head symbol symbols attached_code
token tokens.
Terminals
atom float integer reserved_symbol reserved_word string char var
'->' ':' 'dot'.
Rootsymbol grammar.
Endsymbol '$end'.
grammar -> declaration : '$1'.
grammar -> rule : '$1'.
```

```
declaration -> symbol symbols 'dot': {'$1', '$2'}.
rule -> head '->' symbols attached_code 'dot': {rule, ['$1' | '$3'],
        '$4'}.
head -> symbol : '$1'.
symbols -> symbol : ['$1'].
symbols -> symbol symbols : ['$1' | '$2'].
attached_code -> ':' tokens : {erlang_code, '$2'}.
attached_code -> '$empty' : {erlang_code,
                 [{atom, 0, '$undefined'}]}.
tokens -> token : ['$1'].
tokens -> token tokens : ['$1' | '$2'].
symbol -> var : value_of('$1').
symbol -> atom : value_of('$1').
symbol -> integer : value_of('$1').
symbol -> reserved_word : value_of('$1').
token -> var : '$1'.
token -> atom : '$1'.
token -> float : '$1'.
token -> integer : '$1'.
token -> string : '$1'.
token -> char : '$1'.
token -> reserved_symbol : {value_of('$1'), line_of('$1')}.
token -> reserved_word : {value_of('$1'), line_of('$1')}.
token -> '->' : {'->', line_of('$1')}.
token -> ':' : {':', line_of('$1')}.
Erlang code.
value_of(Token) ->
    element(3, Token).
line_of(Token) ->
    element(2, Token).
```

Note:
The symbols `'->'`, and `':'` have to be treated in a special way, as they are meta symbols of the grammar notation, as well as terminal symbols of the Yecc grammar.

5. The file `erl_parse.yrl` in the `lib/stdlib/src` directory contains the grammar for Erlang.

Note:
Syntactic tests are used in the code associated with some rules, and an error is thrown (and caught by the generated parser to produce an error message) when a test fails. The same effect can be achieved with a call to `return_error(Error_line, Message_string)`, which is defined in the `yeccpre.hrl` default header file.

Files

`lib/parsetools/include/yeccpre.hrl`

See Also

Aho & Johnson: ’LR Parsing’, ACM Computing Surveys, vol. 6:2, 1974.

Index of Modules and Functions

Modules are typed in *this way*.
Functions are typed in `this way`.