Ocamlyacc Tutorial

Ocamlyacc Adaptation: SooHyoung Oh

Ocamlyacc Tutorial

by Ocamlyacc Adaptation: SooHyoung Oh

This is a tutorial on how to use `ocamlyacc` which is distributed with Ocaml language (http://ocaml.org).

Lots of part of this document are borrowed from the bison (https://www.gnu.org/software/bison/) manual.

Please mail all comments and suggestions to <`ohama100 at gmail dot com`>

The latest version can be found at https://ohama.github.io/ocaml/ocamllex-ocamlyacc/ocamlyacc-tutorial/index.html (https://ohama.github.io/ocaml/ocamllex-ocamlyacc/ocamlyacc-tutorial/index.html).

The companion tutorial for `ocamllex` is available at
https://ohama.github.io/ocaml/ocamllex-ocamlyacc/ocamllex-tutorial/index.html
(https://ohama.github.io/ocaml/ocamllex-ocamlyacc/ocamllex-tutorial/index.html).

You can find the source of this document in ocamlyacc-tutorial-src.tar.gz (https://ohama.github.io/ocaml/ocamllex-ocamlyacc/ocamlyacc-tutorial-src.tar.gz). For who wants the other form, pdf (A4 size) (https://ohama.github.io/ocaml/ocamllex-ocamlyacc/ocamlyacc-tutorial.pdf) for printing and html (tar.gz) (https://ohama.github.io/ocaml/ocamllex-ocamlyacc/ocamlyacc-tutorial.tar.gz) are presented.

The source of the examples used in this document can be found ocamlyacc-examples.tar.gz (https://ohama.github.io/ocaml/ocamllex-ocamlyacc/ocamlyacc-examples.tar.gz).

First release: 2004-11-16

Last updated: 2019-05-05

Table of Contents

Chapter 1. Introduction

`Ocamlyacc` is a general-purpose parser generator that converts a grammar description for an LALR(1) context-free grammar into a Ocaml program to parse that grammar. Once you are proficient with Ocamlyacc, you may use it to develop a wide range of language parsers, from those used in simple desk calculators to complex programming languages.

Ocamlyacc is very close to the well-known yacc (or bison) commands that can be found in most C programming environments. Anyone familiar with Yacc should be able to use Ocamlyacc with little trouble. You need to be fluent in Ocaml programming in order to use Ocamlyacc or to understand this manual.

We begin with tutorial chapters that explain the basic concepts of using Ocamlyacc and show three explained examples, each building on the last. If you don't know Ocamlyacc or Yacc, start by reading these chapters. Reference chapters follow which describe specific aspects of Ocamlyacc in detail.

Some explanation is not suitable for the earlier version than 3.08 of Ocaml (Ocamlyacc), in that case, there will be comments like "*(Since Ocaml 3.08)*".

* *All license term in this document is NOT related with ocamlyacc; it is ONLY for this document.*

Chapter 2. The Concepts of Ocamlyacc

This chapter introduces many of the basic concepts without which the details of Ocamlyacc will not make sense. If you do not already know how to use Ocamlyacc, we suggest you start by reading this chapter carefully.

2.1. Languages and Context-Free Grammars

In order for Ocamlyacc to parse a language, it must be described by a *context-free grammar*. This means that you specify one or more *syntactic groupings* and give rules for constructing them from their parts. For example, in the C language, one kind of grouping is called an 'expression'. One rule for making an expression might be, "An expression can be made of a minus sign and another expression". Another would be, "An expression can be an integer". As you can see, rules are often recursive, but there must be at least one rule which leads out of the recursion.

The most common formal system for presenting such rules for humans to read is *Backus-Naur Form* or "BNF", which was developed in order to specify the language Algol 60. Any grammar expressed in BNF is a context-free grammar. The input to Ocamlyacc is essentially machine-readable BNF.

Not all context-free languages can be handled by Ocamlyacc, only those that are LALR(1). In brief, this means that it must be possible to tell how to parse any portion of an input string with just a single token of look-ahead. Strictly speaking, that is a description of an LR(1) grammar, and LALR(1) involves additional restrictions that are hard to explain simply; but it is rare in actual practice to find an LR(1) grammar that fails to be LALR(1). See Mysterious Reduce/Reduce Conflicts, for more information on this.

In the formal grammatical rules for a language, each kind of syntactic unit or grouping is named by a *symbol*. Those which are built by grouping smaller constructs according to grammatical rules are called *nonterminal symbols*; those which can't be subdivided are called *terminal symbols* or *token types*. We call a piece of input corresponding to a single terminal symbol a `token`, and a piece corresponding to a single nonterminal symbol a *grouping*.

We can use the C language as an example of what symbols, terminal and nonterminal, mean. The tokens of C are identifiers, constants (numeric and string), and the various keywords, arithmetic operators and punctuation marks. So the terminal symbols of a grammar for C include 'identifier', 'number', 'string', plus one symbol for each keyword, operator or punctuation mark: 'if', 'return', 'const', 'static', 'int', 'char', 'plus-sign', 'open-brace', 'close-brace', 'comma' and many more. (These tokens can be subdivided into characters, but that is a matter of lexicography, not grammar.)

Here is a simple C function subdivided into tokens:

```
int             /* keyword 'int' */
```

```
square (int x)  /* identifier, open-paren, identifier, identifier, close-paren */
{               /* open-brace */
  return x * x; /* keyword 'return', identifier, asterisk, identifier, semicolon */
}               /* close-brace */
```

The syntactic groupings of C include the expression, the statement, the declaration, and the function definition. These are represented in the grammar of C by nonterminal symbols 'expression', 'statement', 'declaration' and 'function definition'. The full grammar uses dozens of additional language constructs, each with its own nonterminal symbol, in order to express the meanings of these four. The example above is a function definition; it contains one declaration, and one statement. In the statement, each `x` is an expression and so is `x * x`.

Each nonterminal symbol must have grammatical rules showing how it is made out of simpler constructs. For example, one kind of C statement is the `return` statement; this would be described with a grammar rule which reads informally as follows:

```
A 'statement' can be made of a 'return' keyword, an 'expression' and a 'semicolon'.
```

There would be many other rules for 'statement', one for each kind of statement in C.

One nonterminal symbol must be distinguished as the special one which defines a complete utterance in the language. It is called the *start symbol*. In a compiler, this means a complete input program. In the C language, the nonterminal symbol 'sequence of definitions and declarations' plays this role.

For example, `1 + 2` is a valid C expression---a valid part of a C program---but it is not valid as an *entire* C program. In the context-free grammar of C, this follows from the fact that 'expression' is not the start symbol.

The Ocamlyacc parser reads a sequence of tokens as its input, and groups the tokens using the grammar rules. If the input is valid, the end result is that the entire token sequence reduces to a single grouping whose symbol is the grammar's start symbol. If we use a grammar for C, the entire input must be a 'sequence of definitions and declarations'. If not, the parser reports a syntax error.

2.2. From Formal Rules to Ocamlyacc Input

A formal grammar is a mathematical construct. To define the language for Ocamlyacc, you must write a file expressing the grammar in Ocamlyacc syntax: a *Ocamlyacc grammar* file. See Ocamlyacc Grammar Files

A nonterminal symbol in the formal grammar is represented in Ocamlyacc input as an identifier, like an identifier in Ocaml. It is like regular Caml symbol, except that it cannot end with ' (single quote). It should start in lower case, such as `expr`, `stmt` or `declaration`.

The Ocamlyacc representation for a terminal symbol is also called a *token types*. Token typess should be declared in Ocamlyacc Declaration Section and they are added as constructors for the token concrete type. As constructors, they should start with upper case: for example, `Integer`, `Identifier`, `IF` or `RETURN`. The terminal symbol `error` is reserved for error recovery. See Symbols.

The grammar rules also have an expression in Ocamlyacc syntax. For example, here is the Ocamlyacc rule for a C `return` statement.

```
stmt:   RETURN expr SEMICOLON
        ;
```

See Syntax of Grammar Rules.

2.3. Semantic Values

A formal grammar selects tokens only by their classifications: for example, if a rule mentions the terminal symbol ‘integer constant’, it means that *any* integer constant is grammatically valid in that position. The precise value of the constant is irrelevant to how to parse the input: if `x+4` is grammatical then `x+1` or `x+3989` is equally grammatical.

But the precise value is very important for what the input means once it is parsed. A compiler is useless if it fails to distinguish between 4, 1 and 3989 as constants in the program! Therefore, each token in a Ocamlyacc grammar has both a token type and a *semantic value*. See Defining Language Semantics, for details.

The token type is a terminal symbol defined in the grammar, such as `INTEGER`, `IDENTIFIER` or `SEMICOLON`. It tells everything you need to know to decide where the token may validly appear and how to group it with other tokens. The grammar rules know nothing about tokens except their types.

The semantic value has all the rest of the information about the meaning of the token, such as the value of an integer, or the name of an identifier. (A token such as `SEMICOLON` which is just punctuation doesn’t need to have any semantic value.)

For example, an input token might be classified as token type `INTEGER` and have the semantic value 4. Another input token might have the same token type `INTEGER` but value 3989. When a grammar rule says that `INTEGER` is allowed, either of these tokens is acceptable because each is an `INTEGER`. When the parser accepts the token, it keeps track of the token’s semantic value.

Each grouping can also have a semantic value as well as its nonterminal symbol. For example, in a calculator, an expression typically has a semantic value that is a number. In a compiler for a programming language, an expression typically has a semantic value that is a tree structure describing the meaning of the expression.

2.4. Semantic Actions

In order to be useful, a program must do more than parse input; it must also produce some output based on the input. In a Ocamlyacc grammar, a grammar rule can have an *action* made up of Ocaml statements. Each time the parser recognizes a match for that rule, the action is executed. See Actions,

Most of the time, the purpose of an action is to compute the semantic value of the whole construct from the semantic values of its parts. For example, suppose we have a rule which says an expression can be the sum of two expressions. When the parser recognizes such a sum, each of the subexpressions has a semantic value which describes how it was built up. The action for this rule should create a similar sort of value for the newly recognized larger expression.

For example, here is a rule that says an expression can be the sum of two subexpressions:

```
expr: expr PLUS expr   { $1 + $3 }
        ;
```

The action says how to produce the semantic value of the sum expression from the values of the two subexpressions.

2.5. Locations

Many applications, like interpreters or compilers, have to produce verbose and useful error messages. To achieve this, one must be able to keep track of the *textual position*, or *location*, of each syntactic construct. Ocamlyacc provides a mechanism for handling these locations.

Each token has a semantic value. In a similar fashion, each token has an associated location, but the type of locations is the same for all tokens and groupings. Moreover, the output parser is equipped with a data structure for storing locations (see Locations, for more details).

Like semantic values, locations can be reached in actions using functions of the Parsing module.

2.6. Ocamlyacc Output: the Parser File

When you run Ocamlyacc, you give it a Ocamlyacc grammar file as input. The output is a Ocaml source file that parses the language described by the grammar. This file is called a *Ocamlyacc parser*. Keep in mind that the Ocamlyacc utility and the Ocamlyacc parser are two distinct programs: the Ocamlyacc utility is a program whose output is the Ocamlyacc parser that becomes part of your program.

The job of the Ocamlyacc parser is to group tokens into groupings according to the grammar rules---for example, to build identifiers and operators into expressions. As it does this, it runs the actions for the grammar rules it uses.

The tokens come from a function called the *lexical analyzer* that you must supply in some fashion. The Ocamlyacc parser calls the lexical analyzer each time it wants a new token. It doesn't know what is "inside" the tokens (though their semantic values may reflect this). Typically the lexical analyzer makes the tokens by parsing characters of text, but Ocamlyacc does not depend on this. See The Lexical Analyzer Function.

The Ocamlyacc parser file is Ocaml code which defines functions which implements that grammar. Entry functions of the generated Ocaml code are named after the start symbols in grammar file. These functions do not make a complete Ocaml program: you must supply some additional functions. One is the lexical analyzer which should be given as an argument of the parser entry function. Another is an error-reporting function which the parser calls to report an error. In addition, a complete Ocaml program must has to call one (or more) of the generated entry functions or the parser will never run. See Parser Interface.

2.7. Stages in Using Ocamlyacc

The actual language-design process using Ocamlyacc, from grammar specification to a working compiler or interpreter, has these parts:

- Formally specify the grammar in a form recognized by Ocamlyacc (see Ocamlyacc Grammar Files). For each grammatical rule in the language, describe the action that is to be taken when an instance of that rule is recognized. The action is described by a sequence of Ocaml statements.
- Write a lexical analyzer to process input and pass tokens to the parser. The lexical analyzer may be written by hand in Ocaml (see The Lexical Analyzer Function). It could also be produced using `ocamllex`, but the use of `ocamllex` is not discussed in this manual.
- Write a controlling function that calls the Ocamlyacc-produced parser.
- Write error-reporting routines.

To turn this source code as written into a runnable program, you must follow these steps:

- Run Ocamlyacc on the grammar to produce the parser.
- Compile the code output by Ocamlyacc, as well as any other source files.
- Link the object files to produce the finished product.

2.8. The Overall Layout of a Ocamlyacc Grammar

The input file for the Ocamlyacc utility is a `Ocamlyacc grammar file`. The general form of a Ocamlyacc grammar file is as follows:

```
%{
    Header (Ocaml code)
%}
    Ocamlyacc declarations
%%
    Grammar rules
%%
    Trailer (Additional Ocaml code)
```

The `%%`, `%{` and `%}` are punctuation that appears in every Ocamlyacc grammar file to separate the sections.

The *header* may define types, variables and functions used in the actions.

The *Ocamlyacc declarations* declare the names of the terminal and nonterminal symbols, and may also describe operator precedence and the data types of semantic values of various symbols.

The *grammar rules* define how to construct each nonterminal symbol from its parts.

The *Trailer* can contain any Ocaml code you want to use.

Chapter 3. Examples

Now we show and explain three sample programs written using Ocamlyacc: a reverse polish notation calculator, an algebraic (infix) notation calculator, and a multi-function calculator. These examples are simple, but Ocamlyacc grammars for real programming languages are written the same way.

3.1. Reverse Polish Notation Calculator

The first example is that of a simple double-precision *reverse polish notation* calculator (a calculator using postfix operators). This example provides a good starting point, since operator precedence is not an issue. The second example will illustrate how operator precedence is handled.

The source code for this calculator is named `rpcalc.mly`. The `.mly` extension is a convention used for Ocamlyacc input files.

3.1.1. Declarations for "rpcalc"

Here are the Ocaml and Ocamlyacc declarations for the reverse polish notation calculator. By default, comments are enclosed between /* and */ (as in C) except in Ocaml code.

```
/* file: rpcalc.mly */
/* Reverse polish notation calculator. */

%{
open Printf
%}

%token <float> NUM
%token PLUS MINUS MULTIPLY DIVIDE CARET UMINUS
%token NEWLINE

%start input
%type <unit> input

%% /* Grammar rules and actions follow */
```

The header section (see The Header Section) has a code of openning "Printf" module.

The second section, Ocamlyacc declarations, provides information to Ocamlyacc about the token types (see The Ocamlyacc Declarations Section). Each terminal must be declared here. The first terminal symbol is the token type for numeric constants which has a value of `float` The possible arithmetic operators are `PLUS, MINUS, MULTIPLY, DIVIDE`, `CARET` for exponetiation and `UMINUS` for unary

minus. These operator terminals don't have any value with it. The last terminal is `NEWLINE`, a token type for the newline character.

You must give the names of the start symbols and their types, too. In this examples, there is one start symbol named `input` which has type of `unit`. For each start symbol, the parser function with the same name is generated.

3.1.2. Grammar Rules for rpcalc

Here are the grammar rules for the reverse polish notation calculator.

```
input:    /* empty */  { }
        | input line  { }
;

line:     NEWLINE  { }
        | exp NEWLINE  { printf "\t%.10g\n" $1; flush stdout }
;

exp:      NUM   { $1 }
        | exp exp PLUS  { $1 +. $2 }
        | exp exp MINUS  { $1 -. $2 }
        | exp exp MULTIPLY  { $1 *. $2 }
        | exp exp DIVIDE  { $1 /. $2 }
       /* Exponentiation */
        | exp exp CARET  { $1 ** $2 }
       /* Unary minus    */
        | exp UMINUS  { -. $1 }
;
%%
```

The groupings of the rpcalc "language" defined here are the expression (given the name `exp`), the line of input (`line`), and the complete input transcript (`input`). Each of these nonterminal symbols has several alternate rules, joined by the `|` punctuator which is read as "or". The following sections explain what these rules mean.

The semantics of the language is determined by the actions taken when a grouping is recognized. The actions are the Ocaml code that appears inside braces. See Actions.

You must specify these actions in Ocaml, but Ocamlyacc provides the means for passing semantic values between the rules. In each action, the semantic value for the grouping that the rule is going to construct should be given. The semantic values of the components of the rule are referred to as `$1`, `$2`, and so on.

3.1.2.1. Explanation of input

Consider the definition of `input`:

```
input:    /* empty */
        | input line
;
```

This definition reads as follows: “A complete input is either an empty string, or a complete input followed by an input line”. Notice that “complete input” is defined in terms of itself. This definition is said to be *left recursive* since `input` appears always as the leftmost symbol in the sequence. See Recursive Rules.

The first alternative is empty because there are no symbols between the colon and the first `|`; this means that `input` can match an empty string of input (no tokens). We write the rules this way because it is legitimate to type `Ctrl-d` right after you start the calculator. It’s conventional to put an empty alternative first and write the comment `/* empty */` in it.

The second alternate rule (`input line`) handles all nontrivial input. It means, “After reading any number of lines, read one more line if possible.” The left recursion makes this rule into a loop. Since the first alternative matches empty input, the loop can be executed zero or more times.

The parser function `input` continues to process input until a grammatical error is seen or the lexical analyzer says there are no more input tokens; we will arrange for the latter to happen at end of file.

3.1.2.2. Explanation of line

Now consider the definition of `line`:

```
line:     NEWLINE      { }
        | exp NEWLINE  { printf "\t%.10g\n" $1; flush stdout }
;
```

The first alternative is a token which is a newline character; this means that rpcalc accepts a blank line (and ignores it, since there is no action). The second alternative is an expression followed by a newline. This is the alternative that makes rpcalc useful. The semantic value of the `exp` grouping is the value of `$1` because the `exp` in question is the first symbol in the alternative. The action prints this value, which is the result of the computation the user asked for.

As you can see, the semantic value associated with the `line` is `unit`.

3.1.2.3. Explanation of expr

The `exp` grouping has several rules, one for each kind of expression. The first rule handles the simplest expressions: those that are just numbers. The second handles an addition-expression, which looks like two expressions followed by a plus-sign. The third handles subtraction, and so on.

```
exp:   NUM  { $1 }
 | exp exp PLUS { $1 +. $2 }
 | exp exp MINUS { $1 -. $2 }
 ...
 ;
```

We have used `|` to join all the rules for `exp`, but we could equally well have written them separately:

```
exp: NUM  { $1 };
exp: exp exp PLUS { $1 +. $2 };
exp: exp exp MINUS { $1 -. $2 };
 ...
```

All of the rules have actions that compute the value of the expression in terms of the value of its parts. For example, in the rule for addition, `$1` refers to the first component `exp` and `$2` refers to the second one. The third component, `PLUS`, has no meaningful associated semantic value, but if it had one you could refer to it as `$3`. When the parser function recognizes a sum expression using this rule, the sum of the two subexpressions' values is produced as the value of the entire expression. See Actions.

The formatting shown here is the recommended convention, but Ocamlyacc does not require it. You can add or change whitespace as much as you wish. For example, this:

```
exp: NUM { $1 } | exp exp PLUS { $1 +. $2 } | ...
```

means the same thing as this:

```
exp:      NUM  { $1 }
        | exp exp PLUS { $1 + $2 }
        | ...
```

The latter, however, is much more readable.

3.1.3. The rpcalc Lexical Analyzer

The lexical analyzer's job is low-level parsing: converting characters or sequences of characters into tokens. The Ocamlyacc parser gets its tokens by calling the lexical analyzer. See The Lexical Analyzer Function.

Only a simple lexical analyzer is needed for the RPN calculator. This lexical analyzer reads in numbers as `float` and returns them as `NUM` tokens. It recognizes '+', '-', '*', '/', '^', 'n' as operators and returns the corresponding token: ADD, MINUS, MULTIPLY, DIVIDE, CARET and UMINUS. When it meets '\n', the returning token is NEWLINE. Spaces and unknown characers are skipped.

The return value of the lexical analyzer function is a value of the concrete token type. The same text used in Ocamlyacc rules to stand for this token type is also a Ocaml expression for the value for the type. Token type is defined by Ocamlyacc as a constructor of the concrete token type. In this example, therefore, `NUM`, `PLUS`, `...` become values for the lexer function to use.

The semantic value of the token (if it has one) is returned with it. In this example, only `NUM` has a semantic value.

Here is the code for the lexical analyzer:

```
(* file: lexer.mll *)
(* Lexical analyzer returns one of the tokens:
   the token NUM of a floating point number,
   operators (PLUS, MINUS, MULTIPLY, DIVIDE, CARET, UMINUS),
   or NEWLINE.  It skips all blanks and tabs, unknown characters
   and raises End_of_file on EOF. *)
{
  open Rtcalc (* Assumes the parser file is "rtcalc.mly". *)
}
let digit = ['0'-'9']
rule token = parse
  | [' ' '\t'] { token lexbuf }
  | '\n' { NEWLINE }
  | digit+
  | "." digit+
  | digit+ "." digit* as num
  { NUM (float_of_string num) }
  | '+'  { PLUS }
  | '-'  { MINUS }
  | '*'  { MULTIPLY }
  | '/'  { DIVIDE }
  | '^'  { CARET }
  | 'n'  { UMINUS }
  | _  { token lexbuf }
  | eof  { raise End_of_file }
```

3.1.4. The Controlling Function

In keeping with the spirit of this example, the controlling function is kept to the bare minimum. To start the process of parsings, the only requirement is that it call parser function `Parser.input` with two argumenst: lexical analyzer function `Lexer.token` and `lexbuf` of `Lexing.lexbuf` type.

```
(* file: main.ml *)
(* Assumes the parser file is "rtcalc.mly" and the lexer file is "lexer.mll". *)
let main () =
  try
    let lexbuf = Lexing.from_channel stdin in
    while true do
      Rtcalc.input Lexer.token lexbuf
    done
  with End_of_file -> exit 0

let _ = Printexc.print main ()
```

3.1.5. The Error Reporting Routine

When ther parser function detects a syntax error, it calls a function named `parse_error` with the string "syntax error" as argument. The default `parse_error` function does nothing and returns, thus initiating error recovery (see Error Recovery). The user can define a customized `parse_error` function in the header section of the grammar file such as:

```
let parse_error s = (* Called by the parser function on error *)
  print_endline s;
  flush stdout
```

After `parse_error` returns, the Ocamlyacc parser may recover from the error and continue parsing if the grammar contains a suitable error rule (see Error Recovery). Otherwise, the parser aborts by raising the `Parsing.Parse_error` exception. We have not written any error rules in this example, so any invalid input will cause the calculator program to raise exception. This is not clean behavior for a real calculator, but it is adequate for the first example.

3.1.6. Running Ocamlyacc to Make the Parser

Before running Ocamlyacc to produce a parser, we need to decide how to arrange all the source code in source files. For our example, we make three files: `rpcalc.mly` for Ocamlyacc grammar file, `lexer.mll` for Ocamllex input file, `main.ml` which contains main function which calls our parser function.

You can use the following command to convert the parser grammar file into a parser file:

`ocamlyacc` *`file_name`*`.mly`

In this example the file was called `rpcalc.mly` (for “Reverse Polish CALCulator”). Ocamlyacc produces a file named *`file_name`*`.ml`. The file output by Ocamlyacc contains the source code for parser function `input`. The additional functions in the input file (`parse_error`) are copied verbatim to the output.

3.1.7. Compiling the Parser File

Here is how to compile and run the parser file and lexer file:

```
# List files in current directory.
$ ls
.depend  Makefile  lexer.mll  main.ml  rpcalc.mly

# Compile the Ocamlyacc parser.
$ make
ocamlyacc rpcalc.mly
ocamlc -c rpcalc.mli
ocamllex lexer.mll
15 states, 304 transitions, table size 1306 bytes
ocamlc -c lexer.ml
ocamlc -c rpcalc.ml
ocamlc -c main.ml
ocamlc -o rpcalc lexer.cmo rpcalc.cmo main.cmo
rm rpcalc.mli lexer.ml rpcalc.ml

# List files again.
$ ls
./   .depend   lexer.cmo  main.cmi   main.ml   rpcalc.cmi  rpcalc.mly
../  Makefile  lexer.cmi  lexer.mll  main.cmo  rpcalc*    rpcalc.cmo
```

The file `rpcalc` now contains the executable code. Here is an example session using `rpcalc`.

```
$ rpcalc
4 9 +
 13
3 7 + 3 4 5 *+-
 -13
3 7 + 3 4 5 * + - n Note the unary minus, n
 13
5 6 / 4 n +
 -3.166666667
3 4 ^   Exponentiation
 81
^D   End-of-file indicator
$
```

3.2. Infix Notation Calculator: calc

We now modify rpcalc to handle infix operators instead of postfix. Infix notation involves the concept of operator precedence and the need for parentheses nested to arbitrary depth. Here is the Ocamlyacc code for `calc.mly`, an infix desk-top calculator.

```
/* file: calc.mly */
/* Infix notatoin calculator -- calc */
%{
  open Printf
%}

/* Ocamlyacc Declarations */
%token NEWLINE
%token LPAREN RPAREN
%token <float> NUM
%token PLUS MINUS MULTIPLY DIVIDE CARET

%left PLUS MINUS
%left MULTIPLY DIVIDE
%left NEG  /* negation -- unary minus */
%right CARET /* exponentiation */

%start input
%type <unit> input

/* Grammar follows */
%%
input: /* empty */ { }
 | input line { }
;
line: NEWLINE  { }
 | exp NEWLINE { printf "\t%.10g\n" $1; flush stdout }
;
exp: NUM   { $1 }
 | exp PLUS exp  { $1 +. $3 }
 | exp MINUS exp  { $1 -. $3 }
 | exp MULTIPLY exp  { $1 *. $3 }
 | exp DIVIDE exp  { $1 /. $3 }
 | MINUS exp %prec NEG { -. $2 }
 | exp CARET exp  { $1 ** $3 }
 | LPAREN exp RPAREN { $2 }
;

%%
```

There are two important new features shown in this code.

In the second section (Ocamlyacc declarations), `%left` says they are left-associative operators. The declarations `%left` and `%right` (right associativity) is used for the declaration of associativity.

Operator precedence is determined by the line ordering of the declarations; the higher the line number of the declaration (lower on the page or screen), the higher the precedence. Hence, exponentiation has the highest precedence, unary minus (`NEG`) is next, followed by `MULTIPLY` and `DIVIDE`, and so on. See Operator Precedence.

The other important new feature is the `%prec` in the grammar section for the unary minus operator. The `%prec` simply instructs Ocamlyacc that the rule `| MINUS exp` has the same precedence as `NEG`---in this case the next-to-highest. See Context-Dependent Precedence.

Here is a sample run of `calc.mly`:

```
$ calc
4 + 4.5 - (34/(8*3+-3))
 6.880952381
-56 + 2
 -54
3 ^ 2
 9
```

3.3. Simple Error Recovery

Up to this point, this manual has not addressed the issue of *error recovery*---how to continue parsing after the parser detects a syntax error. All we have handled is error reporting with `parse_error`. Recall that by default, the parser function raises exception after calling `parse_error`. This means that an erroneous input line causes the calculator program to raise exception and exit. Now we show how to rectify this deficiency.

The Ocamlyacc language itself includes the reserved word `error`, which may be included in the grammar rules. In the example below it has been added to one of the alternatives for `line`:

```
line:     NEWLINE
        | exp NEWLINE  { printf "\t%.10g\n" $1; flush stdout }
        | error NEWLINE  { }
;
```

This addition to the grammar allows for simple error recovery in the event of a parse error. If an expression that cannot be evaluated is read, the error will be recognized by the third rule for `line`, and parsing will continue. (The `parse_error` function is still called.) The action executes the statement and continues to parse.

This form of error recovery deals with syntax errors. There are other kinds of errors; for example, division by zero, which raises an exception that is normally fatal. A real calculator program must handle this exception and resume parsing input lines; it would also have to discard the rest of the current line of input. We won't discuss this issue further because it is not specific to Ocamlyacc programs.

3.4. Location Tracking Calculator: ltcalc

This example extends the infix notation calculator with location tracking. This feature will be used to improve the error messages. For the sake of clarity, this example is a simple integer calculator, since most of the work needed to use locations will be done in the lexical analyser.

3.4.1. Declarations for ltcalc

The Ocaml and Ocamlyacc declarations for the location tracking calculator are the same as the declarations for the infix notation calculator except `open Lexing`.

```
/* file: ltcalc.mly */
/* Location tracking calculator. */
%{
open Printf
open Lexing
%}

/* Ocamlyacc Declarations */
%token NEWLINE
%token LPAREN RPAREN
%token <float> NUM
%token PLUS MINUS MULTIPLY DIVIDE CARET

%left PLUS MINUS
%left MULTIPLY DIVIDE
%left NEG /* negation -- unary minus */
%right CARET /* exponentiation */

%start input
%type <unit> input

/* Grammar follows */
%%
```

Note there are no declarations specific to locations. Defining a data type for storing locations is not needed: we will use the type provided by default (see Data Type of Locations), which is a four member structure with the following fields:

```
type Lexing.position = {
  pos_fname : string;
  pos_lnum : int;
  pos_bol : int;
  pos_cnum : int;
}
```

3.4.2. Grammar Rules for Itcalc

Whether handling locations or not has no effect on the syntax of your language. Therefore, grammar rules for this example will be very close to those of the previous example: we will only modify them to benefit from the new information.

Here, we will use locations to report divisions by zero, and locate the wrong expressions or subexpressions.

```
input: /* empty */ { }
 | input line { }
;
line: NEWLINE  { }
 | exp NEWLINE { Printf.printf "\t%.10g\n" $1; flush stdout }
;
exp: NUM   { $1 }
 | exp PLUS exp  { $1 +. $3 }
 | exp MINUS exp  { $1 -. $3 }
 | exp MULTIPLY exp  { $1 *. $3 }
 | exp DIVIDE exp  { if $3 <> 0.0 then $1 /. $3
      else (
        let start_pos = Parsing.rhs_start_pos 3 in
        let end_pos = Parsing.rhs_end_pos 3 in
        printf "%d.%d-%d.%d: division by zero"
          start_pos.pos_lnum (start_pos.pos_cnum - start_pos.pos_bol)
          end_pos.pos_lnum (end_pos.pos_cnum - end_pos.pos_bol);
        1.0
      )
    }
 | MINUS exp %prec NEG { -. $2 }
 | exp CARET exp  { $1 ** $3 }
 | LPAREN exp RPAREN { $2 }
;
```

This code shows how to reach locations inside of semantic actions. For rule components, use the following functions,

```
val Parsing.rhs_start_pos : int -> Lexing.position
val Parsing.rhs_end_pos : int -> Lexing.position
```

where the integer parameter says the position of the components on the right-hand side of the rule. It is 1 for the leftmost component.

For groupings, use the following functions.

```
val Parsing.symbol_start_pos : unit -> Lexing.position
val Parsing.symbol_end_pos : unit -> Lexing.position
```

We don't need to calculate the values of the position: the output parser does it automatically. `symbol_start_pos` is set to the beginning of the leftmost component, and `symbol_end_pos` to the end of the rightmost component.

3.4.3. The "ltcalc" Lexical Analyzer

Until now, we relied on Ocamlyacc's defaults to enable location tracking. The next step is to rewrite the lexical analyser, and make it able to feed the parser with the token locations, as it already does for semantic values.

To this end, we must take into account every single character of the input text, to avoid the computed locations of being fuzzy or wrong. `lexbuf.lex_curr_p.pos_cnum` is updated automatically for scanning a character by the lexer engine, so you have to update only `lexbuf.lex_curr_p.pos_lnum` and `lexbuf.lex_curr_p.pos_bol`.

```
(* file: lexer.mll *)
(* Lexical analyzer returns one of the tokens:
   the token NUM of a floating point number,
   operators (PLUS, MINUS, MULTIPLY, DIVIDE, CARET, UMINUS),
   or NEWLINE.  It skips all blanks and tabs, and unknown characters
   and raises End_of_file on EOF. *)

{
  open Ltcalc
  open Lexing
  let incr_lineno lexbuf =
    let pos = lexbuf.lex_curr_p in
    lexbuf.lex_curr_p <- { pos with
      pos_lnum = pos.pos_lnum + 1;
      pos_bol = pos.pos_cnum;
    }
}
let digit = ['0'-'9']
rule token = parse
  | [' ' '\t'] { token lexbuf }
  | '\n'  { incr_lineno lexbuf; NEWLINE }
  | digit+
  | "." digit+
  | digit+ "." digit* as num
  { NUM (float_of_string num) }
  | '+'  { PLUS }
  | '-'  { MINUS }
  | '*'  { MULTIPLY }
  | '/'  { DIVIDE }
  | '^'  { CARET }
  | '('  { LPAREN }
  | ')'  { RPAREN }
  | _  { token lexbuf }
  | eof  { raise End_of_file }
```

Basically, the lexical analyzer performs the same processing as before: it skips blanks and tabs, and reads numbers, operators or delimiters. In addition, it updates `lexbuf.lex_curr_p` containing the token's location.

Now, each time this function returns a token, the parser has its number as well as its semantic value, and its location in the text.

Remember that computing locations is not a matter of syntax. Every character must be associated to a location update, whether it is in valid input, in comments, in literal strings, and so on.

3.5. Multi-Function Calculator: mfcalc

Now that the basics of Ocamlyacc have been discussed, it is time to move on to a more advanced problem. The above calculators provided only five functions, `+`, `-`, `*`, `/` and `^`. It would be nice to have a calculator that provides other mathematical functions such as `sin`, `cos`, etc.

In this example, we will show how to implement built-in functions whose syntax has this form:

function_name (*argument*)

At the same time, we will add memory to the calculator, by allowing you to create named variables, store values in them, and use them later. Here is a sample session with the multi-function calculator:

```
$ mfcalc
pi = 3.14159265
        3.1415927
sin(pi/2)
        1
alpha = beta1 = 2.3
        2.3
alpha
        2.3
log(alpha)
        0.83290912
exp(log(beta1))
        2.3
^D
$
```

Note that multiple assignment and nested function calls are permitted.

3.5.1. Declarations for mfcalc

Here are the Ocaml and Ocamlyacc declarations for the multi-function calculator.

```
/* file: mfcalc.mly */
%{

open Printf
open Lexing

let var_table = Hashtbl.create 16

%}

/* Ocamlyacc Declarations */
%token NEWLINE
%token LPAREN RPAREN EQ
%token <float> NUM
%token PLUS MINUS MULTIPLY DIVIDE CARET
%token <string> VAR
%token <float->float> FNCT

%left PLUS MINUS
%left MULTIPLY DIVIDE
%left NEG /* negation -- unary minus */
%right CARET /* exponentiation */

%start input
%type <unit> input

/* Grammar follows */
%%
```

Since values can have various types, it is necessary to associate a type with each grammar symbol whose semantic value is used. These symbols are `NUM`, `VAR`, `FNCT`, and `exp`. The declarations of terminals are augmented with information about their data type (placed between angle brackets).

The data type of non-terminal, `exp`, is nomally declared implicitly by the action.

In header section, a hash table called `var_table` is created for storing variable's name and value. This will be used in the semantic action part. See The mfcalc Symbol Table.

3.5.2. Grammar Rules for mfcalc

Here are the grammar rules for the multi-function calculator. Most of them are copied directly from `calc`; three rules, those which mention `VAR` or `FNCT`, are new.

```
input: /* empty */ { }
 | input line { }
;
line: NEWLINE  { }
 | exp NEWLINE { printf "\t%.10g\n" $1; flush stdout }
 | error NEWLINE { }
;
exp: NUM   { $1 }
 | VAR   { try Hashtbl.find var_table $1
      with Not_found ->
        printf "no such variable '%s'\n" $1;
        0.0
    }
 | VAR EQ exp  { Hashtbl.replace var_table $1 $3;
      $3
    }
 | FNCT LPAREN exp RPAREN { $1 $3 }
 | exp PLUS exp  { $1 +. $3 }
 | exp MINUS exp  { $1 -. $3 }
 | exp MULTIPLY exp { $1 *. $3 }
 | exp DIVIDE exp { $1 /. $3 }
 | MINUS exp %prec NEG { -. $2 }
 | exp CARET exp  { $1 ** $3 }
 | LPAREN exp RPAREN { $2 }
;

%%
```

For the meaning of the semantic actions related with `VAR`, see The mfcalc Symbol Table.

3.5.3. The mfcalc Symbol Table

The multi-function calculator requires a symbol table to keep track of the names and meanings of variables and functions. This doesn't affect the grammar rules (except for the actions) or the Ocamlyacc declarations, but it requires some additional Ocaml functions for support.

In this example, we use two symbol tables: one for variables and one for functions. The variable symbol table is defined and used in parser and the function symbol table is defined and used in lexer.

The variable symbol table itself is implemented using the hash table. It has a key of `string` type and a value of `float` type.

It is a simple job to modify this code to install predefined variables such as `pi` or `e` as well.

Two important functions allow look-up and installation of symbols in the symbol table. The function `Hashtbl.replace` is passed a name and the value of the variable to be installed. The function

`Hashtbl.find` is passed the name of the symbol to look up. If found, the value of that symbol is returned; otherwise zero is returned.

The lexical analyzer function must now recognize variables, numeric values, and the arithmetic operators. Strings of alphanumeric characters with a leading non-digit are recognized as either variables or functions depending on what the symbol table says about them.

The string is used for look up in the function symbol table. If the name appears in the table, the corresponding function is returned to the parser function as the value of `FNCT`. If it is not, `VAR` with the string is returned.

No change is needed in the handling of numeric values and arithmetic operators in the lexical analyzer function.

```
(* file: lexer.mll *)

{
  open Mfcalc
  open Lexing

  let create_hashtable size init =
    let tbl = Hashtbl.create size in
    List.iter (fun (key, data) -> Hashtbl.add tbl key data) init;
    tbl

  let fun_table = create_hashtable 16 [
    ("sin", sin);
    ("cos", cos);
    ("tan", tan);
    ("asin", asin);
    ("acos", acos);
    ("atan", atan);
    ("log", log);
    ("exp", exp);
    ("sqrt", sqrt);
  ]
}

let digit = ['0'-'9']
let ident = ['a'-'z' 'A'-'Z']
let ident_num = ['a'-'z' 'A'-'Z' '0'-'9']
rule token = parse
  | [' ' '\t'] { token lexbuf }
  | '\n' { NEWLINE }
  | digit+
  | "." digit+
  | digit+ "." digit* as num
  { NUM (float_of_string num) }
  | '+'  { PLUS }
  | '-'  { MINUS }
```

```
| '*'  { MULTIPLY }
| '/'  { DIVIDE }
| '^'  { CARET }
| '('  { LPAREN }
| ')'  { RPAREN }
| '='  { EQ }
| ident ident_num* as word
  { try
    let f = Hashtbl.find fun_table word in
    FNCT f
  with Not_found -> VAR word
  }
| _  { token lexbuf }
| eof  { raise End_of_file }
```

A hash table named `fun_table` is used for storing a function name and a value, that is, a function definition. It is initialized in the header part of the lexer file.

This program is both powerful and flexible. You may easily add new functions.

Chapter 4. Ocamlyacc Grammar Files

Ocamlyacc takes as input a context-free grammar specification and produces a Ocaml-language function that recognizes correct instances of the grammar. The Ocamlyacc grammar input file conventionally has a name ending in `.yml`. See Invoking Ocamlyacc.

4.1. Outline of a Ocamlyacc Grammar

A Ocamlyacc grammar file has four main sections, shown here with the appropriate delimiters:

```
%{
 Header - Ocaml declarations (Ocaml code)
%}
 Ocamlyacc declarations
%%
 Grammar rules
%%
 Trailer - Additional Ocaml code (Ocaml code)
```

By default, comments are enclosed between /* and */ (as in C) except in Ocaml code. So use /* and */ in the *declarations* and *rules* sections, (* and *) in *header* and *trailer* sections.

4.1.1. The Header Section

The *header* section contains declarations of functions and variables that are used in the actions in the grammar rules. These are copied to the beginning of the parser file so that they precede the definition of the *parser function*. You can open the other module in this area. If you don't need any Ocaml declarations, you may omit the `%{` and `%}` delimiters that bracket this section.

4.1.2. The Ocamlyacc Declarations Section

The *ocamlyacc declarations* section contains declarations that define terminal and nonterminal symbols, specify precedence, and so on. At least, there must be one `%start` and the corresponding `%type` directives. See Ocamlyacc declarations.

4.1.3. The Grammar Rules Section

The `grammar rules` section contains one or more Ocamlyacc grammar rules, and nothing else. See Syntax of Grammar Rules.

There must always be at least one grammar rule, and the first `%%` (which precedes the grammar rules) may never be omitted even if it is the first thing in the file.

4.1.4. The Trailer Section

The *trailer* section is copied verbatim to the end of the parser file, just as the *header* section is copied to the beginning. This is the most convenient place to put anything that you want to have in the parser file but which need not come before the definition of the *parse function*. See Parser Interface.

If the last section is empty, you may omit the `%%` that separates it from the grammar rules.

4.2. Symbols, Terminal and Nonterminal

Symbols in Ocamlyacc grammars represent the grammatical classifications of the language.

A *terminal symbol* (also known as a *token type*) represents a class of syntactically equivalent tokens. You use the symbol in grammar rules to mean that a token in that class is allowed. The symbol is represented in the Ocamlyacc parser by a value of variant type, and the *lexer function* function returns a token type to indicate what kind of token has been read.

A *nonterminal symbol* stands for a class of syntactically equivalent groupings. The symbol name is used in writing grammar rules. It should start with lower case.

Symbol names can contain letters, digits (not at the beginning), underscores.

The terminal symbols in the grammar is a *token type* which is a value of variable type in Ocaml. So it should be start with upper case. Each such name must be defined with a Ocamlyacc declaration with `%token`. See Token Type Names.

The value returned by the *lexer function* is always one of the terminal symbols. Each token type becomes a Ocaml value of variant type in the parser file, so *lexer function* can return one.

Because the *lexer function* is defined in a separate file, you need to arrange for the token-type definitions to be available there. After invoking "`ocamlyacc` *filename*`.mly`", the file *filename*`.mli` is generated which contains token-type definitions. It is used in *lexer function*.

The symbol `error` is a terminal symbol reserved for error recovery (see Error Recovery); you shouldn't use it for any other purpose.

4.3. Syntax of Grammar Rules

A Ocamlyacc grammar rule has the following general form:

```
result:
   symbol ... symbol { semantic-action }
 | ...
 | symbol ... symbol { semantic-action }
 ;
```

where *result* is the nonterminal symbol that this rule describes, and *symbol* are various terminal and nonterminal symbols that are put together by this rule (see Symbols).

For example,

```
exp:      exp PLUS exp  {}
        ;
```

says that two groupings of type `exp`, with a `PLUS` token in between, can be combined into a larger grouping of type `exp`.

Whitespace in rules is significant only to separate symbols. You can add extra whitespace as you wish.

At the end of the components there must be one *action* that determine the semantics of the rule. An action looks like this:

```
{Ocaml code}
```

See Actions for detail description.

Multiple rules for the same *result* can be written separately or can be joined with the vertical-bar character `|` as follows:

```
result:
   rule1-symbol ... rule1-symbol { rule1-semantic-action }
 | rule2-symbol ... rule2-symbol { rule2-semantic-action }
 | ...
 ;
```

They are still considered distinct rules even when joined in this way.

If *components* in a rule is empty, it means that *result* can match the empty string. For example, here is how to define a comma-separated sequence of zero or more `exp` groupings:

```
expseq:   /* empty */ {}
```

```
        | expseq1  {}
        ;

expseq1:  exp   {}
        | expseq1 COMMA exp  {}
        ;
```

It is customary to write a comment `/* empty */` in each rule with no components.

4.4. Recursive Rules

A rule is called `recursive` when its *result* nonterminal appears also on its right hand side. Nearly all Ocamlyacc grammars need to use recursion, because that is the only way to define a sequence of any number of a particular thing. Consider this recursive definition of a comma-separated sequence of one or more expressions:

```
expseq1:  exp   {}
        | expseq1 COMMA exp  {}
        ;
```

Since the recursive use of `expseq1` is the leftmost symbol in the right hand side, we call this `left recursion`. By contrast, here the same construct is defined using `right recursion`:

```
expseq1:  exp   {}
        | exp COMMA expseq1  {}
        ;
```

Any kind of sequence can be defined using either left recursion or right recursion, but you should always use left recursion, because it can parse a sequence of any number of elements with bounded stack space. Right recursion uses up space on the Ocamlyacc stack in proportion to the number of elements in the sequence, because all the elements must be shifted onto the stack before the rule can be applied even once. See The Ocamyacc Parser Algorithm, for further explanation of this.

`Indirect` or `mutual` recursion occurs when the result of the rule does not appear directly on its right hand side, but does appear in rules for other nonterminals which do appear on its right hand side.

For example:

```
expr:     primary   {}
        | primary PLUS primary {}
        ;

primary:  constant   {}
        | LPAREN expr RPAREN {}
        ;
```

defines two mutually-recursive nonterminals, since each refers to the other.

4.5. Defining Language Semantics

The grammar rules for a language determine only the syntax. The semantics are determined by the semantic values associated with various tokens and groupings, and by the actions taken when various groupings are recognized.

For example, the calculator calculates properly because the value associated with each expression is the proper number; it adds properly because the action for the grouping `x + y` is to add the numbers associated with *x* and *y*.

4.5.1. Data Types of Semantic Values

In a simple program it may be sufficient to use the same data type for the semantic values of all language constructs. But in most programs, you will need different data types for different kinds of tokens and groupings. For example, a numeric constant may need type `int` or `float`, while a string constant or an identifier might need type `string`.

To use more than one data type for semantic values in one parser, Ocamlyacc requires you to do: Choose one of those types for each symbol (terminal or nonterminal) for which semantic values are used. This is done for tokens with the `%token` Ocamlyacc declaration (see Token Type Names) and for groupings with the `%type` Ocamlyacc declaration (see Nonterminal Symbols).

4.5.2. Actions

An action accompanies a syntactic rule and contains Ocaml code to be executed each time an instance of that rule is recognized. The task of most actions is to compute a semantic value for the grouping built by the rule from the semantic values associated with tokens or smaller groupings.

An action consists of Ocaml statements surrounded by braces. All rules have just one action at the end of the rule, following all the components.

The Ocaml code in an action can refer to the semantic values of the components matched by the rule with the construct `$n`, which stands for the value of the *n*th component. The value of the evaluation of the action is the value for the grouping being constructed.

Here is a typical example:

```
exp:    ...
```

```
        | exp PLUS exp { $1 +. $3 }
```

This rule constructs an `exp` from two smaller `exp` groupings connected by a plus-sign token. In the action, `$1` and `$3` refer to the semantic values of the two component `exp` groupings, which are the first and third symbols on the right hand side of the rule. The sum is returned so that it becomes the semantic value of the addition-expression just recognized by the rule. If there were a useful semantic value associated with the `PLUS` token, it could be referred to as `$2`.

4.6. Tracking Locations

Though grammar rules and semantic actions are enough to write a fully functional parser, it can be useful to process some additionnal informations, especially symbol locations.

The way locations are handled is defined by providing a data type, and actions to take when rules are matched.

4.6.1. Data Type of Locations

* *This content of this section is valid since Ocaml 3.08.*

The data type for locations has the following type:

```
type position = {
   pos_fname : string; (* file name *)
   pos_lnum : int;  (* line number *)
   pos_bol : int;  (* the offset of the beginning of the line *)
   pos_cnum : int;  (* the offset of the position *)
}
```

The value of `pos_bol` field is the number of characters between the beginning of the file and the beginning of the line while the value of `pos_cnum` field is the number of characters between the beginning of the file and the position.

The lexing engine manages only the `pos_cnum` field of `lexbuf.lex_curr_p` with the number of characters read from the start of `lexbuf`. So you are reponsible for the other fields to be accurate. Before using the location in the parser, you have to set `Lexing.lexbuf.lex_curr_p` correctly in lexer, using such a function like this:

```
let incr_linenum lexbuf =
  let pos = lexbuf.Lexing.lex_curr_p in
  lexbuf.Lexing.lex_curr_p <- { pos with
    Lexing.pos_lnum = pos.Lexing.pos_lnum + 1;
    Lexing.pos_bol = pos.Lexing.pos_cnum;
  }
```

```
;;
```

4.6.2. Actions and Locations

Actions are not only useful for defining language semantics, but also for describing the behavior of the output parser with locations. The most obvious way for building locations of syntactic groupings is very similar to the way semantic values are computed. In a given rule, several constructs can be used to access the locations of the elements being matched. The location of the *n*th component of the right hand side can be obtained with:

```
val Parsing.rhs_start : int -> int
val Parsing.rhs_end : int -> int
```

`Parsing.rhs_start n` returns the offset of the first character of the *n*th item on the right-hand side of the rule, while `Parsing.rhs_end n` returns the offset after the last character of the item. are to be called in the action part of a grammar rule only. *n* is 1 for the leftmost item and the first character in a file is at offset 0.

Or you can use the following functions:

```
val Parsing.rhs_start_pos : int -> Lexing.position
val Parsing.rhs_end_pos : int -> Lexing.position
```

(Since Ocaml 3.08) These functions return a position instead of an offset (see Data Type of Locations).

The location of the left hand side grouping can be referred by

```
val Parsing.symbol_start : unit -> int
val Parsing.symbol_end : unit -> int
```

The `symbol_start ()` returns the offset of the first character of the left-hand side of the rule while `symbol_end ()` returns the offset after the last character.

(Since Ocaml 3.08) The following functions are same as `symbol_start` and `symbol_end`, except returning a position instead of an offset (see Data Type of Locations).

```
val Parsing.symbol_start_pos : unit -> Lexing.position
val Parsing.symbol_end_pos : unit -> Lexing.position
```

Here is a basic example using the default data type for locations:

```
exp: ...
 | exp DIVIDE exp
  { if $3 <> 0.0 then $1 /. $3
```

```
    else (
      let start_pos = Parsing.rhs_start_pos 3 in
      let end_pos = Parsing.rhs_end_pos 3 in
      printf "%d.%d-%d.%d: division by zero"
        start_pos.pos_lnum (start_pos.pos_cnum - start_pos.pos_bol)
        end_pos.pos_lnum (end_pos.pos_cnum - end_pos.pos_bol);
      1.0
    )
  }
```

4.7. Ocamlyacc Declarations

The `Ocamlyacc declarations` section of a Ocamlyacc grammar defines the symbols used in formulating the grammar and the data types of semantic values. See Symbols.

All token type must be declared. Nonterminal symbols must be declared if you need to specify which data type to use for the semantic value (see Data Types of Semantic Values).

The first rule in the file also specifies the start symbol, by default. If you want some other symbol to be the start symbol, you must declare it explicitly (see Languages and Context-Free Grammars).

4.7.1. Token Type Names

The basic way to declare a token type name (terminal symbol) is as follows:

```
%token name ... name
%token <type> name ... name
```

Ocamlyacc will convert this into a token type in the parser, so that the lexer function can use the name *name* to stand for this token type's code.

In the event that the token has a value, you must augment the `%token` declaration to include the data type alternative delimited by angle-brackets (see Data Types of Semantic Values).

For example:

```
%token <float> NUM /* define toke NUM and its type */
```

The *type* part is an arbitrary Caml type expression,

* *Because only the <type> part is copied to the `.mli` output file, all type constructor names must be fully qualified (e.g. `Module_name.type_name`) for all types except standard built-in types.*

4.7.2. Operator Precedence

Use the `%left`, `%right` or `%nonassoc` declaration to specify token's precedence and associativity, all at once. These are called `precedence declarations`. See Operator Precedence, for general information on operator precedence.

The syntax of a precedence declaration is

```
%left symbols ...symbols
%right symbols ...symbols
%nonassoc symbols ...symbols
```

They specify the associativity and relative precedence for all the *symbols*:

- The associativity of an operator *op* determines how repeated uses of the operator nest: whether *x op y op z* is parsed by grouping *x* with *y* first or by grouping *y* with *z* first. `%left` specifies left-associativity (grouping *x* with *y* first) and `%right` specifies right-associativity (grouping *y* with *z* first). `%nonassoc` specifies no associativity, which means that *x op y op z* is considered a syntax error.
- The precedence of an operator determines how it nests with other operators. All the tokens declared in a single precedence declaration have equal precedence and nest together according to their associativity. When two tokens declared in different precedence declarations associate, the one declared later has the higher precedence and is grouped first.

4.7.3. Nonterminal Symbols

You can declare the value type of each nonterminal symbol for which values are used. This is done with a `%type` declaration, like this:

```
%type <type> nonterminal ... nonterminal
```

Here *nonterminal* is the name of a nonterminal symbol, and *type* is the name of the type that you want. You can give any number of start nonterminal symbols in the same `%type` declaration, if they have the same value type. Use spaces to separate the symbol names.

This is necessary for start symbols. For the *type* part, see Token Type Names.

4.7.4. The Start-Symbol

You have to declare the start symbols using `%start` declaration as follows:

```
%start symbol ... symbol
```

Each start symbol has a parsing function with the same name in the output file so you can use it as an entry point for the grammar. As noted eariler, type should be assinged to each start symbol using `%type` directive (see Nonterminal Symbols).

4.7.5. Ocamlyacc Declaration Summary

Here is a summary of the declarations used to define a grammar:

- `%token` Declare a terminal symbol (token type name) with no precedence or associativity specified (see Token Type Names).
- `%right` Declare a terminal symbol (token type name) that is right-associative (see Operator Precedence).
- `%left` Declare a terminal symbol (token type name) that is left-associative (see Operator Precedence).
- `%nonassoc` Declare a terminal symbol (token type name) that is nonassociative (using it in a way that would be associative is a syntax error) (see Operator Precedence).
- `%type` Declare the type of semantic values for a nonterminal symbol (see Nonterminal Symbols).
- `%start` Specify the grammar's start symbol (see The Start-Symbol).

Chapter 5. Parser Interface

The Ocamlyacc parser is actually Ocaml functions named after *start symbols* (see The Start-Symbol). Here we describe the interface conventions of *parser functions* and the other functions that it needs to use.

5.1. The Parser Function

To cause parsing to occur, you call the *parser function* with two parameters. The first parameter is the lexical analyzer function of type

```
Lexing.lexbuf -> token
```

and the second is a value of `Lexing.lexbuf` type.

If the start symbol is `parse` in the file `parser.mly` and the lexer function is is `token` of the file `lexer.mll`, the typical usage is:

```
let lexbuf = Lexing.from_channel stdin in
...
let result = Parser.parse Lexer.token lexbuf in
...
```

This parser function reads tokens, executes actions, and ultimately returns when it encounters end-of-input or an unrecoverable syntax error.

5.2. The Lexical Analyzer Function

The *lexical analyzer* function, named after rule declarations, recognizes tokens from the input stream and returns them to the parser. Ocamlyacc does not create this function automatically; you must write it so that *parser function* can call it. The function is sometimes referred to as a lexical scanner.

This function is usually generated by `ocamllex`. See Chapter 12 Lexer and parser generators (ocamllex, ocamlyacc) (http://caml.inria.fr/ocaml/htmlman/manual026.html).

5.3. The Error Reporting Function

The Ocamlyacc parser detects a `parse error` or `syntax error` whenever it reads a token which cannot satisfy any syntax rule. An action in the grammar can also explicitly proclaim an error, using the

`raise Parsing.Parse_error`.

The Ocamlyacc parser expects to report the error by calling an error reporting function named `parse_error`, which is optional. The default `parse_error` function does nothing and returns. It is called by the parser function whenever a syntax error is found, and it receives one argument. For a parse error, the string is normally `"syntax error"`.

The following definition suffices in simple programs:

```
let parse_error s = print_endline s
```

After `parse_error` returns to the *parse function*, the latter will attempt error recovery if you have written suitable error recovery grammar rules (see Error Recovery). If recovery is impossible, the *parse function* will raise `Parsing.Parse_error` exception.

Chapter 6. The Ocamlyacc Parser Algorithm

As Ocamlyacc reads tokens, it pushes them onto a stack along with their semantic values. The stack is called the `parser stack`. Pushing a token is traditionally called `shifting`.

For example, suppose the infix calculator has read `1 + 5 *`, with a `3` to come. The stack will have four elements, one for each token that was shifted.

But the stack does not always have an element for each token read. When the last *n* tokens and groupings shifted match the components of a grammar rule, they can be combined according to that rule. This is called `reduction`. Those tokens and groupings are replaced on the stack by a single grouping whose symbol is the result (left hand side) of that rule. Running the rule's action is part of the process of reduction, because this is what computes the semantic value of the resulting grouping.

For example, if the infix calculator's parser stack contains this:

```
1 + 5 * 3
```

and the next input token is a newline character, then the last three elements can be reduced to 15 via the rule:

```
expr: expr MULTIPLY expr;
```

Then the stack contains just these three elements:

```
1 + 15
```

At this point, another reduction can be made, resulting in the single value 16. Then the newline token can be shifted.

The parser tries, by shifts and reductions, to reduce the entire input down to a single grouping whose symbol is the grammar's start-symbol (see Languages and Context-Free Grammars).

This kind of parser is known in the literature as a bottom-up parser.

6.1. Look-Ahead Tokens

The Ocamlyacc parser does *not* always reduce immediately as soon as the last *n* tokens and groupings match a rule. This is because such a simple strategy is inadequate to handle most languages. Instead, when a reduction is possible, the parser sometimes "looks ahead" at the next token in order to decide what to do.

When a token is read, it is not immediately shifted; first it becomes the `look-ahead token`, which is not on the stack. Now the parser can perform one or more reductions of tokens and groupings on the stack, while the look-ahead token remains off to the side. When no more reductions should take place, the look-ahead token is shifted onto the stack. This does not mean that all possible reductions have been done; depending on the token type of the look-ahead token, some rules may choose to delay their application.

Here is a simple case where look-ahead is needed. These three rules define expressions which contain binary addition operators and postfix unary factorial operators (`FACTORIAL` for '!'), and allow parentheses for grouping.

```
expr:     term PLUS expr
        | term
        ;

term:     LPAREN expr RPAREN
        | term FACTORIAL
        | NUMBER
        ;
```

Suppose that the tokens `1 + 2` have been read and shifted; what should be done? If the following token is `RPAREN`, then the first three tokens must be reduced to form an `expr`. This is the only valid course, because shifting the `RPAREN` would produce a sequence of symbols `term RPAREN`, and no rule allows this.

If the following token is `FACTORIAL`, that is '!', then it must be shifted immediately so that `2 !` can be reduced to make a `term`. If instead the parser were to reduce before shifting, `1 + 2` would become an `expr`. It would then be impossible to shift the `!` because doing so would produce on the stack the sequence of symbols `expr FACTORIAL`. No rule allows that sequence.

6.2. Shift/Reduce Conflicts

Suppose we are parsing a language which has if-then and if-then-else statements, with a pair of rules like this:

```
if_stmt:
          IF expr THEN stmt
        | IF expr THEN stmt ELSE stmt
        ;
```

Here we assume that `IF`, `THEN` and `ELSE` are terminal symbols for specific keyword tokens.

When the `ELSE` token is read and becomes the look-ahead token, the contents of the stack (assuming the input is valid) are just right for reduction by the first rule. But it is also legitimate to shift the `ELSE`, because that would lead to eventual reduction by the second rule.

This situation, where either a shift or a reduction would be valid, is called a `shift/reduce conflict`. Ocamlyacc is designed to resolve these conflicts by choosing to shift, unless otherwise directed by operator precedence declarations. To see the reason for this, let's contrast it with the other alternative.

Since the parser prefers to shift the `ELSE`, the result is to attach the else-clause to the innermost if-statement, making these two inputs equivalent:

```
if x then if y then win (); else lose;

if x then do; if y then win (); else lose; end;
```

But if the parser chose to reduce when possible rather than shift, the result would be to attach the else-clause to the outermost if-statement, making these two inputs equivalent:

```
if x then if y then win (); else lose;

if x then do; if y then win (); end; else lose;
```

The conflict exists because the grammar as written is ambiguous: either parsing of the simple nested if-statement is legitimate. The established convention is that these ambiguities are resolved by attaching the else-clause to the innermost if-statement; this is what Ocamlyacc accomplishes by choosing to shift rather than reduce. (It would ideally be cleaner to write an unambiguous grammar, but that is very hard to do in this case.) This particular ambiguity was first encountered in the specifications of Algol 60 and is called the "dangling `else`" ambiguity.

The definition of `if_stmt` above is solely to blame for the conflict, but the conflict does not actually appear without additional rules. Here is a complete Ocamlyacc input file that actually manifests the conflict:

```
%token IF THEN ELSE variable
%%
stmt:     expr
        | if_stmt
        ;

if_stmt:
          IF expr THEN stmt
        | IF expr THEN stmt ELSE stmt
        ;

expr:     variable
        ;
```

6.3. Operator Precedence

Another situation where shift/reduce conflicts appear is in arithmetic expressions. Here shifting is not always the preferred resolution; the Ocamlyacc declarations for operator precedence allow you to specify when to shift and when to reduce.

6.3.1. When Precedence is Needed

Consider the following ambiguous grammar fragment (ambiguous because the input `1 - 2 * 3` can be parsed in two different ways):

```
expr:     expr MINUS expr
        | expr MULTIPLY expr
        | expr LT expr
        | LPAREN expr RPAREN
        ...
        ;
```

Suppose the parser has seen the tokens `1`, – and `2`; should it reduce them via the rule for the subtraction operator? It depends on the next token. Of course, if the next token is `)`, we must reduce; shifting is invalid because no single rule can reduce the token sequence `- 2 )` or anything starting with that. But if the next token is `*` or `**`, we have a choice: either shifting or reduction would allow the parse to complete, but with different results.

To decide which one Ocamlyacc should do, we must consider the results. If the next operator token *op* is shifted, then it must be reduced first in order to permit another opportunity to reduce the difference. The result is (in effect) `1 - (2 op 3)`. On the other hand, if the subtraction is reduced before shifting *op*, the result is `(1 - 2) op 3`. Clearly, then, the choice of shift or reduce should depend on the relative precedence of the operators – and *op*: `*` should be shifted first, but not `**`.

What about input such as `1 - 2 - 5`; should this be `(1 - 2) - 5` or should it be `1 - (2 - 5)`? For most operators we prefer the former, which is called `left association`. The latter alternative, `right association`, is desirable for assignment operators. The choice of left or right association is a matter of whether the parser chooses to shift or reduce when the stack contains `1 - 2` and the look-ahead token is –: shifting makes right-associativity.

6.3.2. Specifying Operator Precedence

Ocamlyacc allows you to specify these choices with the operator precedence declarations `%left` and `%right`. Each such declaration contains a list of tokens, which are operators whose precedence and associativity is being declared. The `%left` declaration makes all those operators left-associative and the `%right` declaration makes them right-associative. A third alternative is `%nonassoc`, which declares that it is a syntax error to find the same operator twice “in a row”.

The relative precedence of different operators is controlled by the order in which they are declared. The first `%left` or `%right` declaration in the file declares the operators whose precedence is lowest, the next such declaration declares the operators whose precedence is a little higher, and so on.

6.3.3. Precedence Examples

In our example, we would want the following declarations:

```
%left LT
%left MINUS
%left MULTIPLY
```

In a more complete example, which supports other operators as well, we would declare them in groups of equal precedence. For example, `'+'` is declared with `'-'`:

```
%left LT GT EQ NE LE GE
%left PLUS MINUS
%left MULTIPLY DIVIDE
```

(Here `NE` and so on stand for the operators for "not equal" and so on. We assume that these tokens are more than one character long and therefore are represented by names, not character literals.)

6.3.4. How Precedence Works

The first effect of the precedence declarations is to assign precedence levels to the terminal symbols declared. The second effect is to assign precedence levels to certain rules: each rule gets its precedence from the last terminal symbol mentioned in the components. (You can also specify explicitly the precedence of a rule. See Context-Dependent Precedence.)

Finally, the resolution of conflicts works by comparing the precedence of the rule being considered with that of the look-ahead token. If the token's precedence is higher, the choice is to shift. If the rule's precedence is higher, the choice is to reduce. If they have equal precedence, the choice is made based on the associativity of that precedence level. The verbose output file made by `-v` (see Invoking Ocamlyacc) says how each conflict was resolved.

Not all rules and not all tokens have precedence. If either the rule or the look-ahead token has no precedence, then the default is to shift.

6.4. Context-Dependent Precedence

Often the precedence of an operator depends on the context. This sounds outlandish at first, but it is really very common. For example, a minus sign typically has a very high precedence as a unary operator, and a somewhat lower precedence (lower than multiplication) as a binary operator.

The Ocamlyacc precedence declarations, `%left`, `%right` and `%nonassoc`, can only be used once for a given token; so a token has only one precedence declared in this way. For context-dependent precedence, you need to use an additional mechanism: the `%prec` modifier for rules.

The `%prec` modifier declares the precedence of a particular rule by specifying a terminal symbol whose precedence should be used for that rule. It's not necessary for that symbol to appear otherwise in the rule. The modifier's syntax is:

`%prec` *`terminal-symbol`*

and it is written after the components of the rule. Its effect is to assign the rule the precedence of *terminal-symbol*, overriding the precedence that would be deduced for it in the ordinary way. The altered rule precedence then affects how conflicts involving that rule are resolved (see Operator Precedence).

Here is how `%prec` solves the problem of unary minus. First, declare a precedence for a fictitious terminal symbol named `UMINUS`. There are no tokens of this type, but the symbol serves to stand for its precedence:

```
...
%left PLUS MINUS
%left MULTIPLY
%left UMINUS
```

Now the precedence of `UMINUS` can be used in specific rules:

```
exp:    ...
        | exp MINUS exp
        ...
        | MINUS exp %prec UMINUS
```

6.5. Parser States

The function `yyparse` is implemented using a finite-state machine. The values pushed on the parser stack are not simply token type codes; they represent the entire sequence of terminal and nonterminal symbols at or near the top of the stack. The current state collects all the information about previous input which is relevant to deciding what to do next.

Each time a look-ahead token is read, the current parser state together with the type of look-ahead token are looked up in a table. This table entry can say, “Shift the look-ahead token.” In this case, it also specifies the new parser state, which is pushed onto the top of the parser stack. Or it can say, “Reduce using rule number *n*.” This means that a certain number of tokens or groupings are taken off the top of the stack, and replaced by one grouping. In other words, that number of states are popped from the stack, and one new state is pushed.

There is one other alternative: the table can say that the look-ahead token is erroneous in the current state. This causes error processing to begin (see Error Recovery).

6.6. Reduce/Reduce Conflicts

A reduce/reduce conflict occurs if there are two or more rules that apply to the same sequence of input. This usually indicates a serious error in the grammar.

For example, here is an erroneous attempt to define a sequence of zero or more `word` groupings.

```
sequence: /* empty */ { printf "empty sequence\n" }
        | maybeword {}
        | sequence word { printf "added word %s\n" $2 }
        ;

maybeword: /* empty */ { printf "empty maybeword\n" }
        | word  { printf "single word %s\n" $1 }
        ;
```

The error is an ambiguity: there is more than one way to parse a single `word` into a `sequence`. It could be reduced to a `maybeword` and then into a `sequence` via the second rule. Alternatively, nothing-at-all could be reduced into a `sequence` via the first rule, and this could be combined with the `word` using the third rule for `sequence`.

There is also more than one way to reduce nothing-at-all into a `sequence`. This can be done directly via the first rule, or indirectly via `maybeword` and then the second rule.

You might think that this is a distinction without a difference, because it does not change whether any particular input is valid or not. But it does affect which actions are run. One parsing order runs the second rule’s action; the other runs the first rule’s action and the third rule’s action. In this example, the output of the program changes.

Ocamlyacc resolves a reduce/reduce conflict by choosing to use the rule that appears first in the grammar, but it is very risky to rely on this. Every reduce/reduce conflict must be studied and usually eliminated. Here is the proper way to define `sequence`:

```
sequence: /* empty */ { printf "empty sequence\n" }
```

```
        | sequence word  { printf "added word %s\n" $2 }
        ;
```

Here is another common error that yields a reduce/reduce conflict:

```
sequence: /* empty */
        | sequence words
        | sequence redirects
        ;

words:    /* empty */
        | words word
        ;

redirects:/* empty */
        | redirects redirect
        ;
```

The intention here is to define a sequence which can contain either `word` or `redirect` groupings. The individual definitions of `sequence`, `words` and `redirects` are error-free, but the three together make a subtle ambiguity: even an empty input can be parsed in infinitely many ways!

Consider: nothing-at-all could be a `words`. Or it could be two `words` in a row, or three, or any number. It could equally well be a `redirects`, or two, or any number. Or it could be a `words` followed by three `redirects` and another `words`. And so on.

Here are two ways to correct these rules. First, to make it a single level of sequence:

```
sequence: /* empty */
        | sequence word
        | sequence redirect
        ;
```

Second, to prevent either a `words` or a `redirects` from being empty:

```
sequence: /* empty */
        | sequence words
        | sequence redirects
        ;

words:    word
        | words word
        ;

redirects:redirect
        | redirects redirect
        ;
```

6.7. Mysterious Reduce/Reduce Conflicts

Sometimes reduce/reduce conflicts can occur that don't look warranted. Here is an example:

```
%token ID COMMA COLON

%%
def:    param_spec return_spec COMMA
        ;
param_spec:
             type
        |    name_list COLON type
        ;
return_spec:
             type
        |    name COLON type
        ;
type:        ID
        ;
name:        ID
        ;
name_list:
             name
        |    name COMMA name_list
        ;
```

It would seem that this grammar can be parsed with only a single token of look-ahead: when a `param_spec` is being read, an `ID` is a `name` if a comma or colon follows, or a `type` if another `ID` follows. In other words, this grammar is LR(1).

However, Ocamlyacc, like most parser generators, cannot actually handle all LR(1) grammars. In this grammar, two contexts, that after an `ID` at the beginning of a `param_spec` and likewise at the beginning of a `return_spec`, are similar enough that Ocamlyacc assumes they are the same. They appear similar because the same set of rules would be active---the rule for reducing to a `name` and that for reducing to a `type`. Ocamlyacc is unable to determine at that stage of processing that the rules would require different look-ahead tokens in the two contexts, so it makes a single parser state for them both. Combining the two contexts causes a conflict later. In parser terminology, this occurrence means that the grammar is not LALR(1).

In general, it is better to fix deficiencies than to document them. But this particular deficiency is intrinsically hard to fix; parser generators that can handle LR(1) grammars are hard to write and tend to produce parsers that are very large. In practice, Ocamlyacc is more useful as it is now.

When the problem arises, you can often fix it by identifying the two parser states that are being confused, and adding something to make them look distinct. In the above example, adding one rule to `return_spec` as follows makes the problem go away:

```
%token BOGUS
...
%%
...
return_spec:
            type
        |   name COMMA type
        /* This rule is never used.  */
        |   ID BOGUS
        ;
```

This corrects the problem because it introduces the possibility of an additional active rule in the context after the `ID` at the beginning of `return_spec`. This rule is not active in the corresponding context in a `param_spec`, so the two contexts receive distinct parser states. As long as the token `BOGUS` is never generated by `yylex`, the added rule cannot alter the way actual input is parsed.

In this particular example, there is another way to solve the problem: rewrite the rule for `return_spec` to use `ID` directly instead of via `name`. This also causes the two confusing contexts to have different sets of active rules, because the one for `return_spec` activates the altered rule for `return_spec` rather than the one for `name`.

```
param_spec:
            type
        |   name_list COMMA type
        ;
return_spec:
            type
        |   ID COMMA type
        ;
```

Chapter 7. Error Recovery

It is not usually acceptable to have a program terminate on a parse error. For example, a compiler should recover sufficiently to parse the rest of the input file and check it for errors; a calculator should accept another expression.

In a simple interactive command parser where each input is one line, it may be sufficient to have the caller catch the exception and ignore the rest of the input line when that happens (and then call *parse function* again). But this is inadequate for a compiler, because it forgets all the syntactic context leading up to the error. A syntax error deep within a function in the compiler input should not cause the compiler to treat the following line like the beginning of a source file.

You can define how to recover from a syntax error by writing rules to recognize the special token `error`. This is a terminal symbol that is reserved for error handling. The Ocamlyacc parser generates an `error` token whenever a syntax error happens; if you have provided a rule to recognize this token in the current context, the parse can continue.

For example:

```
stmnts:  /* empty string */  {}
        | stmnts NEWLINE  {}
        | stmnts exp NEWLINE {}
        | stmnts error NEWLINE {}
```

The fourth rule in this example says that an error followed by a newline makes a valid addition to any `stmnts`.

What happens if a syntax error occurs in the middle of an `exp`? The error recovery rule, interpreted strictly, applies to the precise sequence of a `stmnts`, an `error` and a newline. If an error occurs in the middle of an `exp`, there will probably be some additional tokens and subexpressions on the stack after the last `stmnts`, and there will be tokens to read before the next newline. So the rule is not applicable in the ordinary way.

But Ocamlyacc can force the situation to fit the rule, by discarding part of the semantic context and part of the input. First it discards states and objects from the stack until it gets back to a state in which the `error` token is acceptable. (This means that the subexpressions already parsed are discarded, back to the last complete `stmnts`.) At this point the `error` token can be shifted. Then, if the old look-ahead token is not acceptable to be shifted next, the parser reads tokens and discards them until it finds a token which is acceptable. In this example, Ocamlyacc reads and discards input until the next newline so that the fourth rule can apply.

The choice of error rules in the grammar is a choice of strategies for error recovery. A simple and useful strategy is simply to skip the rest of the current input line or current statement if an error is detected:

```
stmnt: error SEMICOLON {} /* on error, skip until SEMICOLON is read */
```

It is also useful to recover to the matching close-delimiter of an opening-delimiter that has already been parsed. Otherwise the close-delimiter will probably appear to be unmatched, and generate another, spurious error message:

```
primary:  LPAREN expr RPAREN {}
        | LPAREN error RPAREN {}
        ...
        ;
```

Error recovery strategies are necessarily guesses. When they guess wrong, one syntax error often leads to another. In the above example, the error recovery rule guesses that an error is due to bad input within one `stmnt`. Suppose that instead a spurious semicolon is inserted in the middle of a valid `stmnt`. After the error recovery rule recovers from the first error, another syntax error will be found straightaway, since the text following the spurious semicolon is also an invalid `stmnt`.

To prevent an outpouring of error messages, the parser will output no error message for another syntax error that happens shortly after the first; only after three consecutive input tokens have been successfully shifted will error messages resume.

Chapter 8. Debugging Your Parser

To debug the parser generated by *ocamlyacc*:

- Generate parsing infomation in the file *grammar*`.output` using *-v* option (like "*ocamlyacc -v filneme.mly*"): the information consists of the parsing table and a report on conflicts.
- Set *p* option of the OCAMLRUNPARAM environment variable: for example, execute "*export OCAMLRUNPARAM='p'* " on bash shell.

The parser prints messages about its actions such as shifting a token, reducing a rule.

You can find rule numbers and state numbers mentioned in the messages at the file *grammar*`.output`.

Chapter 9. Invoking Ocamlyacc

The usual way to invoke Bison is as follows:

```
ocamlyacc filename.mly
```

Here *filename*`.mly` is the grammar file name. The parser file's name is made by replacing the `.mly` with `.ml`. Thus, the "`ocamlyacc foo.mly`" yields `foo.ml`.

9.1. Ocamlyacc Options

Here is a list of options that can be used with Ocamlyacc:

- -v

  By default, this option generates a file *grammar*`.output`. It contains parsing infomation such as a description of the parsing tables and a report on ambiguities in the grammar.

- -b*fname*

  Change the name of the output files to *fname*`.ml`, *fname*`.mli` and *fname*`.output`.

Chapter 10. License of This Document

* *All license term in this document is NOT related with ocamlyacc; it is ONLY for this document.*

10.1. Bison License

The bison manual (http://www.gnu.org/software/bison/manual/) requires "GNU General Public License" and "GNU Free Documentation License".

10.1.1. License of bison manual

10.1.2. Conditions for Using Bison

As of Bison version 1.24, we have changed the distribution terms for yyparse to permit using Bison's output in nonfree programs. Formerly, Bison parsers could be used only in programs that were free software.

The other GNU programming tools, such as the GNU C compiler, have never had such a requirement. They could always be used for nonfree software. The reason Bison was different was not due to a special policy decision; it resulted from applying the usual General Public License to all of the Bison source code.

The output of the Bison utility--the Bison parser file--contains a verbatim copy of a sizable piece of Bison, which is the code for the yyparse function. (The actions from your grammar are inserted into this

function at one point, but the rest of the function is not changed.) When we applied the GPL terms to the code for yyparse, the effect was to restrict the use of Bison output to free software.

We didn’t change the terms because of sympathy for people who want to make software proprietary. Software should be free. But we concluded that limiting Bison’s use to free software was doing little to encourage people to make other software free. So we decided to make the practical conditions for using Bison match the practical conditions for using the other GNU tools.

10.1.3. Copying This Manual

GNU Free Documentation License: License for copying bison manual.

10.1.3.1. GNU Free Documentation License

10.1.3.1.1. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

10.1.3.1.2. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools

are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

10.1.3.1.3. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

10.1.3.1.4. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

10.1.3.1.5. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

GNU FDL Modification Conditions

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version’s license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

10.1.3.1.6. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

10.1.3.1.7. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

10.1.3.1.8. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

10.1.3.1.9. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you

also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

10.1.3.1.10. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10.1.3.1.11. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

10.1.3.1.12. ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Sample Invariant Sections list

> Copyright (c) YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

Sample Invariant Sections list

with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

10.1.4. GNU General Public License

Version 2, June 1991

10.1.4.1. Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software - to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation’s software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps:

1. copyright the software, and
2. offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

10.1.4.2. TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

10.1.4.2.1. Section 0

This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program " means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification ".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

10.1.4.2.2. Section 1

You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

10.1.4.2.3. Section 2

You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

1. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.
2. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.
3. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License.

   Exception:: If the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

10.1.4.2.4. Section 3

You may copy and distribute the Program (or a work based on it, under Section 2 in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

1. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
2. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
3. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

10.1.4.2.5. Section 4

You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10.1.4.2.6. Section 5

You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

10.1.4.2.7. Section 6

Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

10.1.4.2.8. Section 7

If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

10.1.4.2.9. Section 8

If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

10.1.4.2.10. Section 9

The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10.1.4.2.11. Section 10

If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

10.1.4.2.12. NO WARRANTY Section 11

BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

10.1.4.2.13. Section 12

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

10.1.4.3. How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

<one line to give the program’s name and a brief idea of what it does.> Copyright (C) <year> <name of author>

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

Gnomovision version 69, Copyright (C) year name of author Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type ‘show w’. This is free software, and you are welcome to redistribute it under certain conditions; type ‘show c’ for details.

The hypothetical commands ‘show w’ and ‘show c’ should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than ‘show w’ and ‘show c’; they could even be mouse-clicks or menu items--whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the program ‘Gnomovision’ (which makes passes at compilers) written by James Hacker.

<signature of Ty Coon>, 1 April 1989 Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

10.2. Ocamlyacc Adaptation Copyright and Permissions Notice

If you are intending to incorporate this document into a published work, please contact the maintainer, and we will make an effort to ensure that you have the most up to date information available.

There is no guarantee that this document lives up to its intended purpose. This is simply provided as a free resource. As such, the authors and maintainers of the information provided within can not make any guarantee that the information is even accurate.