`dypgen` User's Manual

Emmanuel Onzon

November 27, 2011

Overview

`dypgen` is a parser generator for Objective Caml. To use it you need to learn the BNF syntax for grammars which is briefly explained in section 1.2. Its main features are:

- This is a GLR parser. This means it can handle ambiguous grammars and outputs the list of all possible parse trees. Even for non ambiguous grammars, GLR parsing allows to define the grammar in a more natural way. It is possible to extract a definition suitable for the documentation directly from the parser source file.
- Ambiguities can be removed by introducing *priorities* and relations between them. This gives a very natural way to express a grammar (the examples in this documentation illustrate this).
- Grammars are self-extensible, i.e. an action can add new rules to the current grammar. Moreover, the modifications can be local. The new grammar is valid only for a well delimited section of the parsed input.
- `dypgen` provides management of local and global immutable data. The user actions can access it and return it modified. These mechanisms adress the problem posed by global mutable data with GLR parsing and let the user control the scope of the data.

  Modifications of local data are preserved when traveling from right to left in a rule or when going down in the parse tree. Modifications of global data are preserved across the complete traversal of the parse tree.

  These data may be used for instance to do type-checking at parsing time in an elegant and acceptable way. The local data may contain the environment that associates a type to each variable while the global data would contain the substitution over types that is usually produced by unification.
- Pattern matching for symbols in right-hand sides of rules is possible. In particular this allows guarded reductions and to bind names to the arguments of actions.
- `dypgen` allows *early actions*, that are semantic actions performed before the end of a rule.
- It is possible to use dypgen as the lexer generator too. In this case regular expressions that match characters strings are allowed in the right-hand side of grammar rules and it is possible to extend the lexer by introducing such rules.
- The operators `*`, `+` and `?` of regular expressions are available for non terminals and nested rules too.

Contents

1 Introduction to dypgen **3**
- 1.1 The calculator example . . . 3
- 1.2 BNF grammars . . . 5
- 1.3 Priorities . . . 6
- 1.4 Semantical actions . . . 6
- 1.5 The calculator with dypgen lexer generator . . . 7

2 Lexer generators **8**
- 2.1 dypgen lexer generator . . . 8
- 2.2 External lexer generators . . . 13
- 2.3 Using a lexer generator different from ocamllex or dypgen . . . 13
- 2.4 Position of the lexer . . . 14
- 2.5 Extending the lexer . . . 15

3 Resolving ambiguities **15**
- 3.1 Priorities with relations . . . 15
- 3.2 Merge functions and keeping several trees . . . 17
  - 3.2.1 Specific merge functions . . . 17
  - 3.2.2 Example . . . 18
  - 3.2.3 Global and generic merge functions . . . 18
  - 3.2.4 Merging on different global and local data . . . 19
  - 3.2.5 Self-derivable non terminals and empty reductions . . . 20
  - 3.2.6 Merge warning . . . 21
- 3.3 Giving up a reduction . . . 21
- 3.4 Preventing a shift . . . 21

4 Auxiliary data **22**
- 4.1 The fields `global_data` and `local_data` . . . 22
- 4.2 Example with `local_data` . . . 24
- 4.3 Equality between data . . . 25
- 4.4 Example with `global_data` . . . 25
- 4.5 Extending the scope of `local_data` . . . 26

5 User actions and rules **27**
- 5.1 Several actions for a rule . . . 27
- 5.2 Pattern matching on symbols . . . 27
- 5.3 Nested rules . . . 28
- 5.4 Inherited attributes . . . 29
- 5.5 Early actions . . . 30
- 5.6 The operators `*`, `+` and `?` . . . 31
- 5.7 Preventing layout characters to be matched with `!` and `-` . . . 32
- 5.8 Knowing the next lexeme in a user action . . . 32

6 Dynamic extension of the grammar **34**
6.1 Adding rules . . . 34
6.2 The type `obj` . . . 36
6.3 Example . . . 38
6.4 Scope of the changes . . . 39
6.5 Adding new non terminals . . . 40
6.6 Extending dynamically priority data . . . 41

7 Parsing control structures **42**
7.1 The record `dyp` and the type `dypgen_toolbox` . . . 42
7.2 The parser commands list and the type `dyp_action` . . . 43
7.3 `parser_pilot`, `parse`, `lexparse` and `update_pp` . . . 44
7.4 Saving and loading a parser . . . 45

8 Names defined by dypgen **48**
8.1 Types and values defined in the `.ml` file . . . 48
8.2 Types and values defined in the `.mli` file . . . 49
8.3 Adding code to the `.mli` file and at the top of the `.ml` file . . . 50
8.4 No generation of the `.mli` file and other options . . . 50
8.5 Defining the token type in a separate file . . . 51
8.6 The module `Dyp` of the library . . . 51

9 Other features **55**
9.1 Preprocessing with cpp . . . 55
9.2 Generated documentation of the grammar . . . 55
9.3 Information about the parsing . . . 56
9.4 Warnings . . . 57
9.5 Error . . . 58
9.6 Maximum number of constructors and using polymorphic variants . . . 58
9.7 Command-line options for `ocamlc` and include paths . . . 58
9.8 Longest and shortest match for the parser . . . 59
9.9 Building with ocamlbuild . . . 59

10 Demonstration programs **59**

A Acknowledgements **60**

B Migration from `ocamlyacc` **61**

C A complete example of grammar extension **62**

1 Introduction to dypgen

1.1 The calculator example

It is traditional to start the documentation of a parser generator with the calculator exemple. Here we only give the grammar file: `dypgen` takes a file ending with `.dyp` as input and generates a `.ml` file and a `.mli` file.

For the program to be complete, one also need to generate a lexical analyser, which can be done with ocamllex, dypgen or other lexer generators. The complete source for this example lies in the directory `demos/calc_ocamllex` of the distribution.

Here is the file defining the calculator grammar:

```
%token LPAREN RPAREN <int> INT PLUS MINUS TIMES DIV EOL

%relation pi<pt<pp   /* same as  pi<pt pt<pp pi<pp */

%start main

%%

main : expr EOL { $1 }

expr :
  | INT                          { $1 }      pi
  | MINUS expr(=pi)              { -$2 }     pi
  | LPAREN expr RPAREN           { $2 }      pi
  | expr(<=pp) PLUS expr(<pp)    { $1 + $3 } pp
  | expr(<=pp) MINUS expr(<pp)   { $1 - $3 } pp
  | expr(<=pt) TIMES expr(<pt)   { $1 * $3 } pt
  | expr(<=pt) DIV expr(<pt)     { $1 / $3 } pt
```

Let us comment it briefly. More details are available later in this documentation.

- The first two lines starting with `%token` define the *tokens* also called *terminal symbols* of the grammar. The lexical analyser is supposed to transform a character stream into a token stream.

  For instance, the token `PLUS` will probably (this is defined by the lexical analyser) correspond to the character `+` in the input stream.

  On the second line, we also indicate that the `INT` tokens comes with a value of type `int`. They correspond to integer values in the input stream.

- The third line starting with `%relation` defines three priority levels, which intuitively will correspond to atomic expressions (`pi`), products (`pt`) and sums (`pp`).

- The line starting with `%start` gives the entry point of the grammar. This means that one can parse an input stream using the function `Calc_parser.main` which is the function you need to call to actually parse something.

- The characters `%%` tell dypgen where the grammar of the parser begins. They are equivalent to the keyword `%parser`.

- All the other lines define the grammar. The next section will explain how to write grammars. Briefly we remark a BNF grammar (explained below) decorated with the new concept of priority and with semantical actions between curly braces.

1.2 BNF grammars

A BNF grammar is a concise way of defining a language, that is a set of sequences of characters. However, it is traditional and may be more efficient to define a language in to steps: lexical analysis and grammatical analysis.

We will not describe lexical analysis here. We will assume an already defined lexer (for instance using `ocamllex`) which defines some sets of words denoted using capital letters. For instance, in that calculator example above, `PLUS` denotes one word "`+`" while `INT` denoted the set of all words representing an integer.

These set of words defined by the lexer are usually called *tokens* or *terminal symbols*.

Then a BNF grammar, describe the language as a set of sequences of terminal symbols (they sometime have to be separated by *spaces*).

The calculator grammar in this context is

```
expr: INT | MINUS expr | LEFTPAR expr RIGHTPAR
  | expr PLUS expr | expr MINUS expr
  | expr TIMES expr | expr DIV expr
```

This in fact just defines the language `expr` as the smallest language containing `INT` and closed by the following construction rules:

- If $w \in$ `expr` then $-w \in$ `expr` (we assume here that `MINUS` contains only the word `-`, and similar assumptions for the other tokens).
- If $w \in$ `expr` then $(w) \in$ `expr`
- If $w, w' \in$ `expr` then $w+w' \in$ `expr`
- If $w, w' \in$ `expr` then $w-w' \in$ `expr`
- If $w, w' \in$ `expr` then $w*w' \in$ `expr`
- If $w, w' \in$ `expr` then $w/w' \in$ `expr`

In general, a grammar is define by a finite number of non-terminal symbols (the calculator grammar has only one non-terminal: `expr`) and a set of rules describing the elements of each non-terminal symbols. A rule associates to one non-terminal symbol a sequence of symbols (terminal or non-terminal).

Then the languages corresponding to each non-terminals are defined simultaneously as the smallest language satisfying every rules.

Let us consider another example:

```
a: INT | a MINUS b
b: INT | b PLUS a
```

This means that `a` and `b` are the smallest languages containing the integers and such that:

- If $w \in$ `a` and $w' \in$ `b` then $w-w' \in$ `a`
- If $w \in$ `b` and $w' \in$ `a` then $w+w' \in$ `b`

Then, it is easy to see that `0-0+0-0` is in the language `a`, because `0` is both in `a` and `b` which implies that `0-0` is in `a`, from which we deduce that `0+0-0` is in `b` and then, it is easy to conclude. However, `0-0+0-0` is not in `b` (an easy exercise).

1.3 Priorities

The problem with our calculator grammar as written in the previous section is that it is ambiguous and wrong because for instance, there are to way to parse `3-2+1`, one way equivalent to `(3-2)+1` and the other way leading to `3-(2+1)`.

The second way is not the usual way to read this expression and will give a wrong answer when we will compute the value of the expression in the semantical action.

We forget to say that our operator should be associated with the left and also that product and division have priority over addition and subtraction.

To say so, `dypgen` provides priority constant and relation over them. In the case of the calculator, we define three priorities constants: `pi`, `pt` and `pp`. We define the relation `<` by `pi<pt`, `pi<pp` and `pt<pp`.

For each rule, we say to which priority it belongs and for each non terminal in a rule, we give the priority it accepts.

The calculator grammar in this context is

```
expr: INT pi
  | MINUS expr(<=pi) pi
  | LEFTPAR expr RIGHTPAR pi
  | expr(<=pp) PLUS expr(<pp) pp
  | expr(<=pp) MINUS expr(<pp) pp
  | expr(<=pt) TIMES expr(<pt) pt
  | expr(<=pt) DIV expr(<pt) pt
```

Let us comment some rules: in the rule `E: expr(<=pp) PLUS expr(<pp) pp`, we say that an expression produced by this rule will be associated with the priority constant `pp`. We also say that on the left of the `PLUS` token, only an expression of priority less or equal than `pp` can appear while on the right, we are more restrictive since we only accept a priority stricly less that `pp`.

For the rule `E: LEFTPAR expr RIGHTPAR pi`, we associate the smallest priority to the resulting expression and we give no constraint for the expression between parentheses.

More details about priorities will be given in the section 3.1.

1.4 Semantical actions

Now, parsing is not just defining acceptable sequence. One as to produce something from the parsed sequence. This is performed using semantical actions, given after each rule.

An action is a piece of `ocaml` code returning data associated with the parsed sequence, it can be used to build a parse-tree or, as with the calculator, to compute a value. Actions can access the semantics (that is the data associated with) each non-terminal. Terminal symbols also can have semantics associated with them by the lexical analyser.

In the code of an action, we access the semantical data of each symbol in the rule using notation `$1, $2, $3...` (`$3` is the semantics of the third symbol in the rule).

Action must be placed after each rule between curly braces and before the priority of the rule.

Let us look again at the calculator grammar, but with the semantical action added:

```
expr:
  | INT                        { $1 }      pi
  | MINUS expr(<=pi)           { -$2 }     pi
```

```
  | LPAREN expr RPAREN         { $2 }      pi
  | expr(<=pp) PLUS expr(<pp)  { $1 + $3 } pp
  | expr(<=pp) MINUS expr(<pp) { $1 - $3 } pp
  | expr(<=pt) TIMES expr(<pt) { $1 * $3 } pt
  | expr(<=pt) DIV expr(<pt)   { $1 / $3 } pt
```

Here, the actions compute the value of the numerical expression and the example is self explaining.

Remark: a non terminal can accept the empty sequence, by writing no symbol before the opening curly brace of the action.

1.5 The calculator with dypgen lexer generator

The calculator example above assumes that we use an external lexer generator like ocamllex. One can use instead the lexer generator provided by dypgen and define both the lexer and the parser in the .dyp file. Here is a version of the calculator that uses dypgen own lexer generator:

```
%start main
%relation pi<pt<pp
%layout [' ' '\t']

%parser

main: expr "\n" { $1 }

expr:
  | ['0'-'9']+      { int_of_string $1 } pi
  | "-" expr(=pi)             { -$2 }     pi
  | "(" expr ")"              { $2 }      pi
  | expr(<=pp) "+" expr(<pp) { $1 + $3 } pp
  | expr(<=pp) "-" expr(<pp) { $1 - $3 } pp
  | expr(<=pt) "*" expr(<pt) { $1 * $3 } pt
  | expr(<=pt) "/" expr(<pt) { $1 / $3 } pt
```

You can find this example in the directory `demos/calc`. Here are the differences with the previous example:

- The line starting with `%layout` defines which characters are considered to be simple layout and that must be discarded by the lexer. They are defined with a regular expression. Here the lexer discards any space and tabulation character.
- Regular expressions can be written directly on the right-hand sides of grammar rules, allowing to define the lexer and the parser at the same time.

In this simple example no terminal symbol appears anymore. However dypgen lexer generator can produce terminal symbols too. Here is another version of the calculator more similar to the first example but still using dypgen's lexer generator:

```
%start main
%relation pi<pt<pp

%lexer

main lexer =
[' ' '\t'] ->
['0'-'9'] -> INT { int_of_string (Dyp.lexeme lexbuf) }
"+" -> PLUS
"-" -> MINUS
"*" -> TIMES
"/" -> DIV
"(" -> LPAREN
")" -> RPAREN

%parser

main : expr EOL { $1 }

expr :
  | INT                         { $1 }      pi
  | MINUS expr(=pi)             { -$2 }     pi
  | LPAREN expr RPAREN          { $2 }      pi
  | expr(<=pp) PLUS expr(<pp)   { $1 + $3 } pp
  | expr(<=pp) MINUS expr(<pp)  { $1 - $3 } pp
  | expr(<=pt) TIMES expr(<pt)  { $1 * $3 } pt
  | expr(<=pt) DIV expr(<pt)    { $1 / $3 } pt
```

2 Lexer generators

You can use dypgen to generate a lexer for your parser or you can use an external lexer generator like ocamllex or ulex.

2.1 dypgen lexer generator

If you use dypgen to generate your lexer then the entry functions (declared with `%start`) have the following type:

```
?global_data:global_data_type ->
?local_data:local_data_type ->
obj Dyp.dyplexbuf ->
(ast_type * string) list
```

The nature of `global_data` and `local_data` is explained in section 4, `obj Dyp.dyplexbuf` is the type of the lexer buffer (the type `obj` is explained in section 6.2). You make a lexer buffer from a string, an input channel or a function with the following functions of the module `Dyp` of the library:

```
val from_string :
  ('token,'obj,'global_data,'local_data,'lexbuf) parser_pilot ->
  string ->
  'obj dyplexbuf

val from_channel :
  ('token,'obj,'global_data,'local_data,'lexbuf) parser_pilot ->
  in_channel ->
  'obj dyplexbuf

val from_function :
  ('token,'obj,'global_data,'local_data,'lexbuf) parser_pilot ->
  (string -> int -> int) ->
  'obj dyplexbuf
```

These functions are similar to the functions of the same names of the module `Lexing` from the standard library. They expect an argument of type `parser_pilot`, this argument is yielded by a function named `pp` that is generated in the `.ml` file. For instance if you want to parse the string `s` you can use:

```
let lexbuf = Dyp.from_string (Parser.pp ()) s
let result = Parser.main lexbuf
```

assuming your parser is defined in the file `parser.dyp` and the starting non terminal is `main`. The result is of type:

```
  (ast_type * string) list
```

`ast_type` denotes the type of the values yielded by the non terminal `main`, the string is the name of the associated priority. The list contains one couple for each interpretation of the input string. The list contains just one couple if the input is unambiguous. The frame of a lexer definition is as follows:

```
%lexer

let ident = regexp ...

rule aux_lexer_1 [arg1 ... argn] =
  parse regexp { action }
        | ...
        | regexp { action }
and aux_lexer_2 [arg1 ... argn] =
  parse ...
and ...

main lexer =
  regexp -> [Token] [{ action }]
  regexp -> [Token] [{ action }]
  ...
```

This definition takes place before the definition of the parser. It begins with declarations of alias for regular expressions. Then come the definitions of auxiliary lexers, those can be used for instance to parse strings or comments. The syntax of their definitions is similar to definitions for ocamllex. Finally the main lexer is defined after the line

```
main lexer =
```

A line

```
  regexp -> Token { action }
```

means that when `regexp` is matched, the lexer returns the token `Token` with the associated value returned by `action`. The action is optional. The token is optional too. If no token name is written then the text matched by the regular expression is considered to be layout characters and is discarded by the lexer. An action can be stated in this case too for its side-effect. Layout characters can also be declared before the definition of the lexer with the keyword `%layout` followed by the regular expression and an action code inside curly braces if needed. Auxiliary lexers can be called from the user action code of the main lexer and of the auxiliary lexers. They are called with their names followed by the arguments the last one being the lexer buffer. The current lexer buffer is always available inside the action code with the name `lexbuf`. You can use a lexer defined by ocamllex as an auxiliary lexer and call it from a user action. To do this you must convert the lexer buffer to the lexer buffer type of ocamllex with the function:

```
val std_lexbuf : 'obj dyplexbuf -> Lexing.lexbuf
```

of the module `Dyp`. For instance say you want to match strings with a lexer `string` defined in a file `lex_string.mll`, you can use the following entry in your main lexer definition:

```
'"' -> STRING { Buffer.clear Lex_string.string_buf;
      Lex_string.string (Dyp.std_lexbuf lexbuf);
      Buffer.contents Lex_string.string_buf }
```

assuming the lexer `string` stores the matched string in the buffer `string_buf` defined in the file `lex_string.mll`.

When using dypgen lexer generator you can write regular expressions directly in the right-hand sides of the production rules of the grammar of the parser. They return a value of type `string` that is available in action code in the corresponding variable like `$1`. A sequence of several regular expressions yields as many variables. For instance in the action of the rule:

```
nt: "hello" "world"
```

the variable `$1` has the value `"hello"` and `$2` the value `"world"`. Any layout character can be matched between `"hello"` and `"world"`. If you enclose the sequence inside parenthesis then it is considered as one regular expression. In the action of the rule:

```
nt: ("hello" "world")
```

the variable `$1` has the value `"helloworld"`. The string `"helloworld"` must be matched to be able to reduce with this rule.

Regular expressions have the same syntax as those of ocamllex except that the operator `#` of character set difference is not available and it is not possible to do binding to sub-patterns (this is done with the keyword `as` in ocamllex).

As with ocamllex the lexer always yields a unique token when there is at least one match. A lexer generated by dypgen deals with ambiguity a bit differently than ocamllex does. Here is how dypgen chooses the matching of the lexer: The matches that are not expected, taking into account what has been parsed so far, are discarded. Among those which are expected the longest are selected, then those belonging to the most recent lexer (if the lexer has been extended), then the one generated by the higher regular expression in the precedence order. The order in which the regular expressions appear in the file is their precedence order. The precedence of regular expressions in right-hand side of grammar rule is unspecified but lower than that of the regular expressions defined in the section `main lexer =` except for those that are just a string: they are of higher precedence instead. The lexer can be extended when new grammar rules containing regular expressions are introduced. In this case the precedence of these new regular expressions follows the same rule. The precedence of regular expressions that match layout characters is lower than that of other regular expressions (including those that match less characters) and is left unspecified among them. Contrary to ocamllex it is not possible to select the shortest match instead of the longest.

You can also keep all the tokens of maximum length (and which are expected). To do so define the following variable:

```
let dypgen_choose_token = `all
```

in the header of the parser definition. And when you call the function `lexparse` (see section 7.3 for more information about this function) use the optional argument:

```
~choose_token:`all
```

Note that in this case if there are several entries for the same terminal symbol in the lexer that are matched then this terminal symbol is yielded as many times with its corresponding action each time. Suppose these lines are in the lexer:

```
['0'-'9']+ -> INT { int_of_string (Dyp.lexeme lexbuf) }
['0'-'9']+ -> INT { -int_of_string (Dyp.lexeme lexbuf) }
```

Then when the lexer reads the integer `17` it returns to the lexer two tokens: `INT(17)` and `INT(-17)`.

Note that when using `` `all `` the behaviour of the lexer pertaining to layout regular expressions stays the same as the default one. If the lexer matches several layout regular expressions (all of maximal length), then there are two possible cases:

- Only layout regular expressions are matched. Then one action is chosen among them and executed.
- In addition to layout regular expressions, some non-layout regular expression is matched. Then no action associated to layout regular expression is executed.

Remark that even when returning several tokens the lexer only returns the tokens of the maximal length and drop the ones with a shorter length. This has counter-intuitive consequences. For example consider the following grammar :

```
{ let dypgen_choose_token = ‘all }
%start main

%lexer
main lexer =
  ['a'-'z']+ -> ID

%parser

main:
  | 'a'? d { "ab" }
  | ID 'c'   { "id" }

d: 'b' { }
```

Parsing fails on the string `b` even with

```
let dypgen_choose_token = ‘all
```

When parsing `b` the lexer first matches `'a'?` with the empty string, then it matches `ID` with the string `b` and the match with `'a'?` is thrown because it is of shorter length. Afterwards the rule `main: 'a'? b` is disqualified.

A nested rule can solve this problem. Change the first rule with this one:

```
main:
  | ["a"]? d { "ab" }
```

and the parsing of `b` does not fail anymore. For more informations about nested rules see the section 5.3.

When parsing a file you want to be able to track the number of lines and maybe from which file the code you are parsing originates before possible preprocessing. To do so the two following functions are available:

```
val set_newline : 'obj dyplexbuf -> unit
val set_fname : 'obj dyplexbuf -> string -> unit
```

These functions update the fields `pos_lnum` and `pos_bol`, and `pos_fname` of the lexer buffer respectively. If `lexbuf` is your lexer buffer, you can access these fields with:

```
(Dyp.std_lexbuf lexbuf).lex_curr_p.pos_lnum
(Dyp.std_lexbuf lexbuf).lex_curr_p.pos_bol
(Dyp.std_lexbuf lexbuf).lex_curr_p.pos_fname
```

2.2 External lexer generators

If you use an external lexer generator like ocamllex to generate your lexer then the entry functions (declared with `%start`) have the following type:

```
?global_data:global_data_type ->
?local_data:local_data_type ->
(lexbuf_type -> token) ->
lexbuf_type ->
(ast_type * string) list
```

Compared with the use of dypgen as a lexer generator, the function expects one extra argument: a function of type:

```
lexbuf_type -> token
```

This function is the lexer that produces tokens for the parser. The type `lexbuf_type` denotes the type of the lexer buffer (with ocamllex it is `Lexing.lexbuf`) and `token` is a type declared by dypgen. The type `token` owns one constructor for each token that is declared with the keyword `%token`.

Note that you cannot use regular expressions in the right-hand side of the rules of the parser for lexing purpose if you do not use dypgen as lexer generator. More precisely: if you write a regular expression in a rhs then dypgen will assume that you use dypgen as lexer generator too.

2.3 Using a lexer generator different from ocamllex or dypgen

If you want to use another lexer than ocamllex or dypgen then you have to define the function `dypgen_lexbuf_position` in the header. If you don't need the position of the lexer you may use the line:

```
let dypgen_lexbuf_position = Dyp.dummy_lexbuf_position
```

The function `dummy_lexbuf_position` is:

```
let dummy_lexbuf_position _ = (Lexing.dummy_pos, Lexing.dummy_pos)
```

With this function you don't have access to relevant information about the position of the lexer while parsing. If you need the position of the lexer in your action code, then you have to define the function `dypgen_lexbuf_position` in the header accordingly (see 2.4).

If the type of the lexer buffer is constructed with some types defined in the header of the parser definition, then you must include them in the `.mli` (see section 8.3).

The demo program `tinyML_ulex` uses Ulex as its lexer generator.
The demo program `position_parser_token_list` uses ocamllex but a different lexer buffer than `Lexing.lexbuf`.

2.4 Position of the lexer

The following functions allow to know the location of the part of the input that is reduced to the symbols that appear in the rule currently applied. They are available from the action code.

```
val symbol_start : unit -> int
val symbol_start_pos : unit -> Lexing.position
val symbol_end : unit -> int
val symbol_end_pos : unit -> Lexing.position
val rhs_start : int -> int
val rhs_start_pos : int -> Lexing.position
val rhs_end : int -> int
val rhs_end_pos : int -> Lexing.position
```

These functions tell what is the part of the input that is reduced to a given non terminal. They should behave the same way as the functions of the same names of the module `Parsing` do in `ocamlyacc`. These functions are part of the record `dyp`. When you use them you must use the record `dyp`, like:

```
dyp.symbol_start ()
```

The demo program `position` illustrates the use of these functions with dypgen as lexer generator, and the program `position_ocamllex` with ocamllex.

If you want these functions to return relevant information when you use a lexer generator different from ocamllex or dypgen, or when you do not use the interface of ocamllex directly with dypgen, then you have to define the function `dypgen_lexbuf_position` in the header of the parser accordingly. The type of this function is:

```
val dypgen_lexbuf_position : lexbuf_type -> (Lexing.position * Lexing.position)
```

where `lexbuf_type` is the type of the lexer buffer (it is infered by Caml). The two positions are the position of the beginning of the last token that has been read by the parser and its end. The type `Lexing.position` is:

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

The demo program `position_token_list` is the same as `position` except that it first uses ocamllex to make a list of tokens and then uses dypgen to parse this list of tokens. It makes use of `dypgen_lexbuf_position`.

Note: the following abbreviations are available in the `.dyp` file: `$<` for `dyp.Dyp.rhs_start_pos` and `$>` for `dyp.Dyp.rhs_end_pos`.

2.5 Extending the lexer

It is possible to extend indirectly the main lexer generated by dypgen. The way to do it is to introduce new grammar rules for the *parser* with regular expressions in the right-hand sides. It is not possible to extend the definition of an existing terminal symbol neither to create a new terminal symbol at runtime. However this does not prevent you from introducing new tokens. If you want a new token at runtime then you can introduce a new rule with a new non terminal in the left-hand side. This new non terminal will represent the new token. The right-hand side of the new rule contains the regular expression that matches the lexemes for the new token. The section 6 explains how to extend the grammar at runtime.

3 Resolving ambiguities

There are two main mechanisms to handle ambiguities in `dypgen`: a system of priorities with relations between them and the merge functions which can decide which parse-tree to keep when a given part of the input is reduced to the same non terminal by two different ways. Two other secondary mechanisms make possible to decide to give up a reduction with a rule (by raising the exception `Dyp.Giveup`) or to prevent a shift (i.e. the parser is prevented from reading more input without performing a reduction).

3.1 Priorities with relations

Each time a reduction by a rule happens, the corresponding parse-tree is yielded with a value which is called a priority. Priorities are named and declared along with relations between them which hold true with the keyword `%relation`. The symbol for the relation is `<` (but this does not mean that it has to be an order). A declaration of priorities can be for instance:

```
%relation pi<pt pt<pp pi<pp
```

It is possible to declare a relation which is transitive on a subset of the priorities with a more compact syntax.

```
%relation p1<p2<...<pn
```

means that the following relations hold: `p1<p2`, ... , `p1<pn`, `p2<p3`, ... , `p2<pn`, ... , `p(n-1)<pn`. Thus the first example of relations declaration can also be written:

```
%relation pi<pt<pp
```

The declarations can use several lines, the following declaration is valid:

```
%relation pi<pt
pt<pp
%relation pi<pp
```

Each rule in the grammar returns a priority value when it is used to reduce. This priority is stated by the user after the action code. For instance:

```
expr: INT { $1 } pi
```

If the parser reduces with this rule then it returns the value associated with the token `INT` and the priority `pi`. The user can state no priority, then the default priority `default_priority` is returned each time a reduction with this rule happens. The value `default_priority` is part of the module `Dyp_priority_data` available in the parser code.

Each non terminal in the right-hand side of a rule is associated with a set of priorities that it accepts to perform a reduction. This set of priorities is denoted using the following symbols `<`, `>` and `=` and a priority `p`.

`(<p)` denotes the set of all priorities `q` such that `q<p` holds. `(<=p)` denotes the previous set to which the priority `p` is added. `(>p)` is the set of all priorities `q` such that `p<q` holds. Obviously `(>=p)` denotes the previous set to which the priority `p` is added and `(=p)` is the set of just `p`. Note that when declaring relations between priorities, the symbols `>` and `=` cannot be used.

If no set of priorities is stated after a non terminal in a right-hand side of a rule, then it means that it accepts any priority. Thus to not state any set of priority is equivalent to state the set of all priorities.

A basic example, you have the following rules:

```
expr: INT { $1 } pi
expr: MINUS expr(<=pi) { -$2 }
```

You parse the string: '`-1`'. First `1` is reduced to `expr` with the first rule. `1` is the returned value and `pi` is the returned priority. Then the reduction with the second rule can happen because the non terminal `expr` in its right-hand side accepts the priority `pi`. This reduction is performed, the returned value is `-1` and the returned priority is `default_priority`. Now if we want to parse the string '`--1`' a syntax error will happen, because when `-1` is reduced, the priority `default_priority` is yielded, which is not accepted by the second rule and therefore a reduction by this rule cannot happen a second time.

Another example, we have the relations `pi<pt<pp` and the following grammar:

```
expr:
  | INT                          { $1 }      pi
  | LPAREN expr RPAREN           { $2 }      pi
  | expr(<=pp) PLUS expr(<pp)    { $1 + $3 } pp
  | expr(<=pt) TIMES expr(<pt)   { $1 * $3 } pt
```

and we parse the following input: `1 + 2 * 3`

`1` and `2` are reduced to `expr`, each returning the priority `pi`, then the parser explores the shift and the reduction. The parser can reduce with the third rule because `pi<pp` holds, the priority `pp` is returned. After `3` is reduced to `expr`, the parser cannot reduce with rule 4 after the reduction by rule 3 because `pp<pt` does not hold. But the exploration of the possibility of a shift of `*` after reducing `2` to `expr` leads to a successful parsing (which respects the precedence of `*` over `+`).

The user can declare a priority without stating any relation with it by adding it in the section after `%relation`. You should declare any prorities which has no relation. If you use a priority that

is not declared and has no relation, in a rule, then `dypgen` will emit a warning.

3.2 Merge functions and keeping several trees

3.2.1 Specific merge functions

When a part of the input is reduced by several different ways to a same non terminal `nt` then the parser must decide whether to keep all the abstract syntax trees (ASTs) yielded or to choose just one or a few of them, or to make a new tree from them. By default only one tree is kept, and the other ones are discarded. It is not possible to know which one is kept. The user can override the default choice. It is possible to define a function `dyp_merge_Obj_nt` of type:

```
val dyp_merge_Obj_nt:
  (nt_type * global_data_t * local_data_t) list ->
  nt_type list * global_data_t * local_data_t
```

The type names stated here are not actually defined anywhere but are used here to make understand the nature of the argument and result of the merge functions. The type `nt_type` represents the type of the value returned when one reduces to the non terminal `nt`. The argument of the function is the list of the ASTs which are the different interpretations which were yielded and kept for the non terminal `nt` for a given part of the input. Theses ASTs are associated with their corresponding `global_data` and `local_data` in a triple (see section 4 for more information about `global_data` and `local_data`). The default behavior of dypgen is to not merge when the data are different. Therefore by default all the global data are actually the same as well as all the local data. This behavior can be made different (see 3.2.4).

The merge function returns a result like:

```
(tree_list, global_data, local_data)
```

Where `tree_list` is a list of ASTs which are kept as the different interpretations for the non terminal `nt` for the considered part of the input. `global_data` and `local_data` are the `global_data` and `local_data` that are kept for the parsing.

The user can define such a merge function for each non terminal in the header of the parser.

For example the following merge function keeps all the ASTs for the non terminal `nt`, without considering `global_data` and `local_data`:

```
let dyp_merge_Obj_nt l = match l with
  | (_,gd,ld)::_ -> List.map (fun (o,_,_) -> o) l, gd, ld
  | [] -> assert false
```

A merge function is only called on ASTs which were yielded with the same priority. If a part of the input is reduced to the same non terminal by two different ways but yielding two distinct priorities, then each AST is kept and used independently for the parsing, but they are not merged.

3.2.2 Example

Here is an example of using a merge function to enforce the precedence of the multiplication over the addition. Suppose we have the following grammar:

```
expr:
  | INT             { Int $1 }
  | expr PLUS expr  { Plus ($1,$2) }
  | expr TIMES expr { Times ($1,$2) }
```

And the following string: `3+10*7` should be parsed `Plus (3,Times (10,7))`, more generally the parse result of any string should respect the precedence of `*` over `+`. Then we can do this by defining the following merge function:

```
let rec dyp_merge_Obj_expr = function
  | (o1,gd,ld)::(o2,_,_)::t ->
  begin match o1 with
    | Times ((Plus _),_)
    | Times (_,(Plus _)) -> dyp_merge_Obj_expr ((o2,gd,ld)::t)
    | _ -> dyp_merge_Obj_expr ((o1,gd,ld)::t)
  end
  | [o,gd,ld] -> [o],gd,ld
  | _ -> assert false
```

You can find this example implemented in the directory `demos/merge_times_plus`.

Note that the argument of the merge function (the list) always has at least two elements when called by dypgen.

Since it is not possible to know which tree is kept by the default merge function, you should only rely on it when all the trees yielded by the ambiguous non terminal are actually the same, or when it does not matter which one is chosen.

3.2.3 Global and generic merge functions

In addition to these merge functions which are specific to one non terminal, the user can also define one global merge function called `dyp_merge`, and several generic merge functions. A generic merge function can be the merge function of several different non terminals. The type of the global merge function is the following:

```
type 'a merge_function =
  ('a * global_data_t * local_data_t) list ->
  'a list * global_data_t * local_data_t
```

The global merge function can be defined in the header of the parser definition, for instance to define a global merge function which keeps all the parse trees:

```
let dyp_merge l = match l with
  | (_,gd,ld)::_ -> List.map (fun (o,_,_) -> o) l, gd, ld
  | [] -> assert false
```

Then it will be the merge function of any non terminal `nt` which has not its own function `dyp_merge_nt` and has no generic merge function assigned to it.

Generic merge functions are defined in the header. Then to assign a merge function to one or several non terminal one uses the keyword `%merge` in the following way:

```
%merge my_merge_function Obj_nt1 Obj_nt2 Obj_nt3 Obj_nt4 Obj_nt5
```

where `my_merge_function` is the name of the generic merge function which has been defined in the header and `nt1` ... `nt5` are the names of the non terminal which are assigned this generic merge function.

Note that the global merge function must be polymorphic and accept the type of values yielded by any non terminal. This is not the case for a generic merge function as long as it is used for non terminals that return values of the same type. In the special case when all the non terminals return values of the same type, the global merge function does not have to be polymorphic.

There are two predefined generic merge functions available to the user in the module `Dyp` of the library:

```
val keep_all : 'a merge_function
val keep_one : 'a merge_function
```

They keep respectively all the ASTs, and one of the AST, they choose `global_data` and `local_data` arbitrarily. If no global merge function is defined then by default it is `keep_one`.

Note that you can use the predefined merge functions as generic functions and as the global merge function as well. If you want to keep all the ASTs for all the non terminals, just define the following in the header:

```
let dyp_merge = keep_all
```

You can find a very simple example using `keep_all` in the directory `demos/forest` where a parse forest is yielded.

Note that the merge functions are in fact associated with the constructors of the type `obj` (see section 6.2) rather than with the non terminals. Which means that any non terminal that shares the same constructor is assigned the same specific merge function. And that using `%merge` you can assign a generic merge function to constructors introduced with the keyword `%constructor` (see section 6.2 for more information). Now keep in mind that despite the fact merge is associated with a constructor, merge is only applied to different parsings of the *same non terminal* (and same part of the input and yielding the same priority) and never to distinct non terminals even if they have the same constructor.

3.2.4 Merging on different global and local data

You may want to merge ASTs even if their corresponding global data or local data are different. Then you have to define the variable:

```
val dypgen_keep_data : [`both|`global|`local|`none]
```

in the header of the parser, or use the functions `parse` or `lexparse` (discussed in section 7.3) with the optional argument:

```
?keep_data : [‘both|‘global|‘local|‘none]
```

- `‘both` is the default behavior: if global data is different or local data is different then the ASTs are not merged.
- `‘global` allows merging only when global data are equal but local data may be different.
- `‘local` allows merging only when local data are equal but global data may be different.
- `‘none` makes merging happen disregarding whether the data are equal or not.

3.2.5 Self-derivable non terminals and empty reductions

In the general case, when a merge function is called for a non terminal `nt` and a given part of the input, it means that this part will never be reduced to `nt` again. The arguments passed to the merge function are all the ASTs that are possible when reducing this part of the input to `nt`.

As a special exception this is not true in two cases:

- When the part of the input that is considered is the empty string
- When the non terminal can derive itself. For instance with the rules:

  ```
  a: b c
  b: a
  c:
  ```

  the non terminal `a` can derive itself (this is also true for `b`).

In such cases, as soon as two possible ASTs are yielded for a part of the input and a given non terminal, the corresponding merge function is called on them. If afterward a third AST is yielded then the merge function is called again on this third AST and on the result of the previous call to the merge function.

In the general case, the reductions are performed in an order such that an AST is never merged *after* being used by another reduction. This ensures, in particular, that a reduction that spans a bigger part of the input does not miss the results of reductions of subparts of the input when they are relevant.

However it is not always possible to avoid this situation when reducing to a self-derivable non terminal. In such a case, a possible solution is to use a reference on a list of all possible ASTs for the nodes corresponding to self-derivable non terminals, and when a merge happen, to update the list.

3.2.6 Merge warning

To know whether a merge happens you can use the command line option `--merge-warning` with `dypgen`. Then the generated parser will emit a warning on the standard output each time a merge is added to its merge working list and print the name of the non terminal which will be merged. The number of merge warnings is equal to the length of the list passed in argument to the merge function minus one.

Warning: when there is an error of type with the arguments or the result of a merge function, Caml reports it and points to the `.ml` file generated by `dypgen`, not to the `.dyp` input file, which may be puzzling. In particular make sure your merge functions return a list of ASTs as the first value of the returned tuple and not just an AST, otherwise you will have an error message pointing to the generated `.ml` file.

3.3 Giving up a reduction

When a reduction occurs this reduction is given up if the exception `Dyp.Giveup` is raised in the corresponding action code.

```
expr:
  | INT           { $1 }
  | expr DIV expr { if $3=0 then raise Dyp.Giveup else ($1 / $3) }
```

This is an example where a division by zero is not syntaxically correct, the parser refuses to reduce a division by zero. We can also imagine a language with input files being parsed and typed at the same time and an action would give up a reduction if it detected an incompatibility of type. Let us assume we have the following input for such a language:

```
exception Exception
let head l = match l with
  | x::_ -> x
  | _ -> raise Exception
let a = head 1::[2]
```

Then the parser tries to reduce `head 1` to an expression, but the typing concludes to an incompatibility of type. Hence an exception `Giveup` is raised which tells the parser not to explore this possibility any further. Then the parser reduces `1::[2]` to an expression and thereafter `head 1::[2]` is reduced to an expression without type error.

3.4 Preventing a shift

When a reduction occurs it is possible to prevent the shift with the action code. You have to state the character '`@`' before the left brace which begins the action code and returns the following list along with the returned value of your action in a couple:

```
@{ returned_value, [Dyp.Dont_shift] }
```

The list after `returned_value` is called the parser commands list, see section 7.2 for more information about it. Here is an example, we have the following rules in a grammar:

```
expr:
  | INT { Int $1 }
  | expr COMMA expr { action_comma $1 $3 }
```

Assuming we have the following input: `1,2,3`, there are two ways to reduce it to `expr`. First one: `1,2` is reduced to `expr` then `expr,3` is reduced to `expr`. Second one: `2,3` is reduced to `expr` then `1,expr` is reduced to `expr`. Now if we have the input `1,2,...,n` there are as many ways to reduce it to `expr` as there are binary trees with n leaves. But we can use the following action code instead:

```
expr:
  | INT { Int $1 }
  | expr COMMA expr @{ action_comma $1 $3, [Dyp.Dont_shift] }
```

Then there is only one way to reduce `1,2,3` to `expr`: the first one, because when `[Dyp.Dont_shift]` is returned the parser will not shift the comma without reducing. And there is only one way to reduce `1,...,n` to `expr`.

The constructor `Dont_shift` is part of the type `dyp_action` (see section 7.2), which is defined in the module `Dyp` of the library.

4 Auxiliary data

4.1 The fields `global_data` and `local_data`

With GLR, the parsing can follow different interpretations independently and simultaneously if there are local ambiguities. As a consequence if there are accesses and changes to data through side-effects for each of these parsings, there can be unwanted interactions between them. For this reason, using side effects for the purpose of storing data should be avoided during the parsing. If you want to build and store data while parsing and access this data from within the action code then you should use immutable data and the record fields `dyp.global_data` or `dyp.local_data`. The record `dyp` is available in any user action code, its type is defined in the module `Dyp` of the library `dyp.cm[x]a`, see the section 7.1 for more information about it. If `dyp.global_data` or `dyp.local_data` are used, then the user must define their initial values in the header with the names:

```
let global_data = some_initial_data
let local_data = some_other_data
```

`global_data` and `local_data` can be passed to the parser as argument since they are optional arguments of the parser. For instance if '`main`' is a start symbol of the grammar and it returns values of type `main_type` then the following function is defined in the generated code:

```
val main :
  ?global_data:gd_type ->
  ?local_data:ld_type ->
  (lexbuf_type -> token) ->
  lexbuf_type ->
  (main_type * string) list
```

where `gd_type` and `ld_type` represents the type of the global and local data. `lexbuf_type` is the type of the lexing buffer, if you use ocamllex then it is `Lexing.lexbuf`.

If you use dypgen to generate your lexer then the type of `main` is:

```
val main :
  ?global_data:global_data_type ->
  ?local_data:local_data_type ->
  obj Dyp.dyplexbuf ->
  (main_type * string) list
```

The record `dyp` is available in the action code and you can access the values of `global_data` and `local_data` through the fields of `dyp` that have the corresponding names. You can return new values for `global_data` and `local_data` using the list of values of type `dyp_action`. To do that you state the character '`@`' before the action code and you return the list

```
[Dyp.Global_data new_global_data]
```

along with the returned value of your action if you want to change `global_data`. Or the list

```
[Dyp.Local_data new_local_data]
```

if you want to change `local_data`. Or the list

```
[Dyp.Global_data new_global_data; Dyp.Local_data new_local_data]
```

if you want to change both `global_data` and `local_data`. For more information about the type of the constructors `Global_data` and `Local_data` and the parser commands list see section 7.2.

The data accessible with `dyp.global_data` follows the parsing during the reductions and the shifts. If at one point the parser follows different alternatives then it evolves independently for each alternative (but side-effects may cause unwanted interactions between them, therefore they should be avoided).

The same is true for `dyp.local_data` except that when a reduction happens, it is 'forgotten' and returned to its previous value. More precisely: when you return a new value for `local_data` in an action which yields a non terminal `nt`, then this `local_data` is not forgotten (unless you do it) in any action which follows until this non terminal `nt` is used in another reduction. When this happens, `dyp.local_data` is forgotten and returns to its previous value just before the execution of the action code of this reduction.

Here is an example:

```
a: TOK_1 b TOK_2 c TOK_3 d EOF @{ action_8, [Local_data some_ld] }
b: e f   @{ action_3, [Local_data local_data_3] }
e: A1    @{ action_1, [Local_data local_data_1] }
f: A2    @{ action_2, [Local_data local_data_2] }
c: g h   @{ action_6, [Local_data local_data_6] }
g: B1    @{ action_4, [Local_data local_data_4] }
h: B2    @{ action_5, [Local_data local_data_5] }
d: C     @{ action_7, [Local_data local_data_7] }
```

We parse the string:

`TOK_1 A1 A2 TOK_2 B1 B2 TOK_3 C EOF}`

- Assume that at the beginning of the parsing `dyp.local_data` has some initial value `local_data_0`.
- The first action to be performed is `action_1`, it has access to `local_data_0` and it returns `local_data_1`,
- then the next action is `action_2`, it has access to `local_data_1` and returns `local_data_2`, although this is useless in this case because it is about to be forgotten.
- Then the reduction of `e f` to `b` happens. `dyp.local_data` comes back to its value before `action_1` was performed, that is `local_data_0`. `action_3` is performed and it returns the value `local_data_3` for `local_data`.
- The next action to be performed is `action_4`, `dyp.local_data` has the value `local_data_3` and it returns `local_data_4`,
- then `action_5` has access to this new value and returns `local_data_5` but it is useless in this case.
- The reduction of `g h` to `c` happens and the value of `dyp.local_data` is again `local_data_3`, the value it had just after `action_3` was applied. `action_6` returns the value `local_data_6` for `local_data`.
- The next action is `action_7` which has access to `local_data_6`. It returns `local_data_7` but it is useless since it is about to be forgotten.
- The reduction with the first rule is performed, the value of `dyp.local_data` comes back to `local_data_0` and the last action `action_8` is performed.

4.2 Example with `local_data`

`dyp.local_data` is useful, for instance, to build a symbol table, and makes possible to use it to disambiguate. Assume the following grammar and action code:

```
main: expr "\n" { $1 }

expr: ['0'-'9']+           { Int (int_of_string $1) }
  | IDENT                  { if is_bound dyp.local_data $1 then Ident $1
                             else raise Giveup }
  | "(" expr ")"           { $2 }
  | expr "+" expr          { Plus ($1,$3) }
  | "let" binding "in" expr { Let ($2,$4) }

binding: IDENT "=" expr
  @{ Binding ($1,$3),
    [Local_data (insert_binding dyp.local_data $1 $3)] }
```

If we keep all the parse trees (see merge functions section 3.2), then the following input string:

```
let x = 1 in x+1
```

yields the two following parse trees:

```
(let x = 1 in (x+1))
((let x = 1 in x)+1)
```

But this input string:

```
let x = 1 in 1+x
```

only yields one parse-tree:

```
(let x = 1 in (1+x))
```

Moreover some input are detected as invalid because of unbound identifiers before the whole string has been parsed, like:

```
(let x = 1 in y+2) + ...
```

This example is available in the directory `demos/local_data`.

4.3 Equality between data

A function of equality between two global data named `global_data_equal` can be defined otherwise it is by default the physical equality `(==)`, same thing for `local_data` but the function is called `local_data_equal`. These equality functions are used by the parser for optimization purpose. In some cases two distinct GLR parsings may share their `global_data` and `local_data` if both of them are deemed equal by `global_data_equal` and `local_data_equal` respectively. In this case one the two data is chosen and the other is discarded for each of the `global_data` and `local_data`.

4.4 Example with `global_data`

Here is a very simple example of use of `dyp.global_data`, it counts the number of reductions.

```
{ open Dyp
let global_data = 0
let global_data_equal = (=) }

%start main
%relation pi<pt<pp
%layout [' ' '\t']

%%

main: expr "\n"
  { Printf.printf "The parser made %d reductions for this interpretation.\n"
      dyp.global_data;
      $1 }
```

```
expr:
  | ['0'-'9']+
      @{ int_of_string $1, [Global_data (dyp.global_data+1)] } pi
  | expr(<=pp) "+" expr(<pp)
      @{ $1 + $3, [Global_data (dyp.global_data+1)] } pp
  | expr(<=pt) "*" expr(<pt)
      @{ $1 * $3, [Global_data (dyp.global_data+1)] } pt
```

For instance we parse `5*7+3*4`, when `4` is reduced, `global_data` is incremented from 4 to 5 (note that it will actually not count the real total number of reductions, but only the number of reductions made for the first interpretation of the input). This example is available in the directory `demos/global_data`.

Here is a less basic example and where `dyp.global_data` can be useful, suppose we have the following grammar:

```
expr:
  | LIDENT
  | INT
  | FLOAT
  | LPAREN expr COMMA expr RPAREN
  | expr PLUS expr
  | LPAREN expr RPAREN
  | expr PLUSDOT expr
```

We parse an input and the following string is a part of this input:

```
(x+.3.14,(x+1,(x+5, ... )))
```

Where `...` stands for something long. Suppose we are typing the expressions in parallel with their parsing and we want to reject the previous string as early as possible. We do not want to wait for reducing the whole string to detect the type incompatibility of `x`. Then we can use `dyp.global_data` for that purpose and when reducing `x+.3.14` we store in `global_data` that `x` is of type float and then when we reduce `x+1` we have this information still stored in `global_data` which is accessible from the action code. And we can detect the type incompatibility whithout having to parse more input.

4.5 Extending the scope of `local_data`

In addition to `dyp.local_data` a field `dyp.last_local_data` is available. It contains the value of `local_data` when the reduction of the last non terminal of the right-hand side of the current rule occured. This makes possible to extend the scope of `local_data` using:

```
[Dyp.Local_data dyp.last_local_data]
```

Extensions of the scope of `local_data` are useful when you want to use a left-recursive rule instead of a right-recursive one and you still want to use `local_data` but not `global_data`. For instance:

```
rev_ident_list:
  |   { [] }
  | rev_ident_list IDENT
      @{ $2::$1, [Dyp.Local_data (String_set.add $2 dyp.last_local_data)] }

ident_list:
  | rev_ident_list @{ List.rev $1, [Dyp.Local_data dyp.last_local_data] }
```

Left-recursive rules are more efficient than right-recursive ones with a GLR parser driven by an LR(0) automaton, which is currently what dypgen generates.

5 User actions and rules

5.1 Several actions for a rule

It is possible to bind several actions to the same rule, but only one will be completely performed. When there are several actions for the same rule, the parser tries them one after the other until the first one that does not raise `Giveup`. To bind several actions to a rule, write the rule as many times as you need actions and state a different action after the rule each time. The actions are tried by the parser in the same order as they appear in the definition file `.dyp`. For instance with the following actions:

```
expr:
  | INT  { if $1 = 0 then raise Dyp.Giveup else $1 }
  | INT  { 100 }
```

an integer returns its value except for `0` which returns `100`.

When a rule is dynamically added to the grammar, if it was already in the grammar, then the action it was bound to is still in the grammar but when a reduction by the rule occurs the old action will be applied only if the new one raises `Giveup`.

It is possible to execute all the actions bound to each rule instead of just the first that does not raise Giveup. For this use the option: `--use-all-actions` or declare:

```
let dypgen_use_all_actions = true
```

in the header of the parser definition.

5.2 Pattern matching on symbols

It is possible to match the value returned by any symbol in a right-hand side against a pattern. The syntax is just to add the pattern inside `<` and `>` after the symbol name and after the possible priority constraint. For instance we may have the following:

```
expr: INT<x>  { x }
```

which is identical to:

```
expr: INT  { $1 }
```

One can use pattern matching to have a guarded reduction. Assuming the lexer return `OP("+")` on '+' and `OP("*")` on '*', we can have the following rules:

```
expr:
  | expr OP<"+"> expr { $1 + $2 }
  | expr OP<"*"> expr { $1 * $2 }
```

The patterns can be any Caml patterns (but without the keyword `when`). For instance this is possible:

```
expr: expr<(Function([arg1;arg2],f_body)) as f> expr  { some action }
```

The value returned by an early action can also be matched. You can write:

```
nt0: TOK1 nt1 ...{ early_action }<pattern> nt2 TOK2 nt3 { action }
```

One can use a pattern after a regular expression too (the value is always of type `string`).

The directory `calc_pattern` contains a version of the calculator which uses patterns (in a basic way).

5.3 Nested rules

A non terminal in a right-hand side can be replaced by a list of right-hand sides in brackets, like:

```
nt1:
| symb1 [  symb2 symb3 { action1 } prio1
         | symb4 symb5 { action2 } prio2
         | symb6 symb7 { action3 } prio3 ]<pattern> symb8 symb9 { action4 } prio4
| ...
```

The pattern in `<` and `>` after the list of nested rules is optional.

In addition to be a convenient way to embed subrules, nested rules allow a useful trick when you use dypgen as the lexer generator. Consider the following grammar :

```
{ let dypgen_choose_token = `all }
%start main

%lexer
main lexer =
  ['a'-'z']+ -> ID

%parser

main:
  | 'a'? d { "ab" }
  | ID 'c'   { "id" }

d: 'b' { }
```

Parsing fails on the string `b` for reasons explained at the end of section 2.1.

A nested rule can solve this problem by changing the first rule with:

```
main:
  | ["a" { }]? d { "ab" }
```

then the parsing of `b` does not fail anymore. You can also omit the action and write it:

```
main:
  | ["a"]? d { "ab" }
```

When an action is omitted then by default the reduction by the rule returns the value bound to the last symbol of the right-hand-side (however it is not allowed to omit the action when the right-hand-side is empty). Note that something like: `['a']` will still be interpreted as a character set and not as a nested rule.

5.4 Inherited attributes

Inherited attributes are a way to send information down the parse tree in a way that is more explicit than with the global and local data discussed in section 4 and that is specific to a non terminal. It can be seen as a parameterization of a non terminal with a Caml value. Here is a simple example:

```
main: a{1} "b" { $1 }
a: "a" { $0 }
```

on the input `ab` the parser returns `1`. Here `{1}` states that the caml integer `1` is sent to the next rule with left-hand side `a`. The symbol `$0` in the action of the second rule is bound to the *inherited attribute* which is `1` here.
Here is an example with one more level:

```
main: a{1} "c" { $1 }
a: b{$0+1} "b" { $1 }
b: "a"         { $0 }
```

On the input `abc` it returns `2`. A version with pattern matching on left-hand side:

```
main: a{1} "c"   { $1 }
a<i>: b{i+1} "b" { $1 }
b<j>: "a"        { j }
```

Inherited attributes are experimental currently, and the full class of L-attribute grammar is not quite supported yet. Left-recursive rules compute inherited attributes for a depth of 2 at most. For example with the following grammar:

```
main: a{1} "b"   { $1 }
a<i>:
  | a{i+1} "a" { $1 }
  | "a"        { i }
```

the input `ab` returns 1 and the input `aab` returns 2, as expected. But the input `aaab` and any input with more than 2 `a` returns the integer 2, and not the number of `a`.

5.5 Early actions

It is possible to insert action code in the right-hand side of a rule before the end (the final action) of this right-hand side. This is an early action. For instance you may have:

```
expr: LET IDENT EQUAL expr ...@{ binding dyp $2 $4 } IN expr { Let ($5, $7) }
```

Note that you have to write three dots '`...`' before the early action. The value returned by the early action is accessible to the final action as if the early action were a symbol in the right-hand side. In the rule above it is accessible with `$5`. The record `dyp` is passed to `binding` to have access to `dyp.local_data`. You may use the character '`@`' depending whether you want to return a parser commands list or not.

For the parser the rule:

```
expr: LET IDENT EQUAL expr IN expr
```

does not exist. Instead dypgen creates the following rule:

```
expr: LET IDENT EQUAL expr dypgen__early_action_0 IN expr
```

Where `dypgen__early_action_0` is a non terminal symbol created by dypgen that sends down inherited attributes that carry the information of the preceding non terminals. Therefore writing

```
expr: LET IDENT EQUAL expr ...@{ binding dyp $2 $4 } IN expr { Let ($5, $7) }
```

is equivalent to:

```
expr: LET IDENT EQUAL expr a{$2, $4} IN expr { Let ($5, $7) }
a<ident, exp>: { binding dyp ident exp }
```

As a consequence if you return a value for `local_data` in the early action, then this new value of `local_data` is accessible to any action which is performed before the final action is applied (the final action not included). For instance if we have:

```
expr: LET IDENT EQUAL expr ...@{ binding dyp $2 $4 } IN expr { Let ($5,$7) }
```

with

```
let binding dyp name exp =
  Binding (name,exp),
  [Local_data (insert_binding dyp.local_data name exp)]
```

Then during the recognition of the last non terminal `expr` of the righ-hand side of the rule, any action has access to the value of `dyp.local_data` that is the value for `local_data` returned by `binding` called by the early action. But it is not accessible anymore when `{ Let ($5,$7) }` is executed.

It is possible to insert several early actions in a rule, like:

```
nt1: TOK1 nt2 ...@{ pa_1 } nt2 nt3 ...{ pa_2 } nt4 TOK2 ...{ pa_3 } TOK3
     { final_action }
```

Note that if a rule exists two times in the parser definition, one time with early actions and another time whithout early action, or both times with early actions, then each of them may be tried by the parser to reduce regardless of whether the other raised `Giveup` or not. This differs from the case where both of them do not have early action where at most one of the actions assigned to the same rule is executed entirely.

Note: if your parser makes use of at least one early action that returns a parser commands list (i.e. of the form `...@{ pa }`), then you must compile with the option `-rectypes` and use the option `--ocaml "-rectypes"` with dypgen (see the end of section 7.2).

5.6 The operators *, + and ?

When a symbol on a right-hand side of a rule is followed by '`*`' then the parser must match nothing or several times this symbol. And the result is a list. For instance the following rule:

```
list: "(" expr * ")"  { $2 }
```

matches a list of zero or more `expr` inside parentheses and `$2` has the type `expr list` assuming `expr` is the type of the values returned when the parser reduces to the non terminal `expr`.

Nested rules can also be followed by `*`. Example of a rule:

```
tuple: "(" expr ["," expr {$2}]* ")"  { $2::$3 }
```

This can also be written without the action `{$2}` since when an action is omitted then by default the reduction by the rule returns the value bound to the last symbol of the right-hand-side (however it is not allowed to omit the action when the right-hand-side is empty):

```
tuple: "(" expr ["," expr]* ")"  { $2::$3 }
```

The operator '`+`' has the same effect as `*` except that the symbol must be matched at least once. Example of a rule:

```
expr: "fun" arg + "->" expr
```

When a symbol is followed by the operator '`?`' it is matched zero or one time. The returned type is an option. The operators `*`, `+` and `?` can be followed by a pattern (see section 5.2 for more information about pattern matching on symbols). After `*` and `+` a pattern must match a list. But after a `?` it does not have to match an option since a pattern `pat` stated after `?` is automatically replaced with: `(Some pat|None)`. For instance you can use: `TOKEN?<"|">` if you want to match either nothing or a terminal named `TOKEN` with the associated value `"|"`. Note that a pattern following a `?` must not contain any binding.

Here are other examples of rules:

```
statement: infix INT symbolchar+ ["," (symbolchar+)]* ";" ";"
      @{ let gd = infix_op ($3::$4) $2 $1 dyp.global_data in
         Infix_stm, [Global_data gd] }

expr: "match" expr "with" "|"? match ["|" match]*
      { Match_with ($2,$5::$6) }
```

These rules are used in the demo program `tinyML`.

5.7 Preventing layout characters to be matched with ! and -

When you use dypgen as the lexer generator you can make a rule match only when no layout character has been matched in the part of the input that is reduced with the rule. To do this state the character '!' at the beginning of the right-hand side of the rule. It has no effect when using another lexer generator than dypgen.

You can use the character '-' before a symbol to forbid layout characters just before the symbol. When stated at the end of a right-hand side before the user action it forbids layout characters before the next token to be read after the reduction by this rule.

Note that when some part of the input is reduced to the same non terminal by two different ways but that in one case layout characters are allowed to follow and in the other case they are not, no merge happen (see section 3.2 for more information about merge functions).

5.8 Knowing the next lexeme in a user action

You may want to know the next lexeme to be matched by the lexer in a user action. If you are using a lexer generated by dypgen then you can do this calling the function `dyp.next_lexeme`:

```
next_lexeme : unit -> string list
```

`next_lexeme` is a field of the record `dyp` (see section 7.1 for more information about this record).

When `dyp.next_lexeme` is called in a user action it returns the list of the next lexemes matched by the parser until the first non-layout lexeme. The list is in reverse order, i.e. the first element of the list is the non-layout lexeme. If the lexer does not match anything then an empty list is returned. If one or several layout lexemes are matched but no non-layout lexeme then a list with the layout lexemes is returned, with the last match first.

Note however that `next_lexeme` does not always exactly returns the lexeme the lexer will read. These discrepancies have two causes. First, `next_lexeme` does not take into account the preceding context as the lexer does, because at this stage `next_lexeme` does not have enough information to do so. Second, when matching the regular expressions of the lexer, `next_lexeme` does not execute the associated user actions, in particular auxiliary lexers are not called. Theses cases are detailed below.

The non-layout lexeme returned by `next_lexeme` might be longer than the one that will be matched by the lexer, i.e. the lexeme read by the lexer is a prefix of the lexeme returned by `next_lexeme`. This case may happen when the lexer can recognize two tokens, one being a prefix of the other, and the context prevents the longer to be matched. An example of this is the following grammar:

```
%start main
%parser
main: nt "b" "c" "d" { () }
nt: "a"   { print_endline (List.hd (dyp.Dyp.next_lexeme ())) }
  | "bc"  { () }
```

Assuming one calls the parser on the string `"abcd"`, `next_lexeme` returns `["bc"]` because the lexer has the regular expression matching the string `"bc"` and when the user action is called the string `"bc"` is a prefix of the remaining of the character stream (i.e. `"bcd"`), and the string `"bc"` is chosen over the string `"b"` because it is longer. Whereas the lexer will only read the character `b` because it does not expect the string `"bc"` given what has already been parsed.

Another case where there is a discrepancy between `next_lexeme` and the lexer is when `next_lexeme` matches a non-layout lexeme whereas the lexer only matches a layout lexeme. Here is an example:

```
%start main
%layout ' '
%parser
main: nt "bcd" { () }
nt: "a"        { print_endline (List.hd (dyp.Dyp.next_lexeme ())) }
  | " b"       { () }
```

Calling the parser on `"a bcd"`, `next_lexeme` returns `[" b"]`, whereas the lexer will first read the layout lexeme `' '`, skip it and then will read the lexeme `"bcd"`.

Finally here are two cases where `next_lexeme` returns a wrong result because it does not call the auxiliary lexer:

```
{ open Dyp
let string_buf = Buffer.create 10 }
%start main
%lexer
rule string = parse
  | '"' { () }
  | _ { Buffer.add_string string_buf (Dyp.lexeme lexbuf); string lexbuf }

main lexer =
'"' -> STRING { Buffer.clear string_buf; string lexbuf;
      Buffer.contents string_buf }

%parser
main: nt STRING { () }
nt: "a" { print_endline (List.hd (dyp.next_lexeme ())) }
```

Calling the lexer on `a"hello"`, `next_lexeme` just returns `["\""]` (a string containing the character '"'), whereas the lexer will read the string `"hello"` ('"' included).

Calling an auxiliary lexer to parse commentaries may have this effect too:

```
%start main
%lexer
rule comment n = parse
  | "*/" { if n>0 then comment (n-1) lexbuf }
  | "/*" { comment (n+1) lexbuf }
  | _ { comment n lexbuf }
```

```
main lexer = "/*" -> { comment 0 lexbuf }

%parser
main: nt "bcd" { () }
nt: "a" { print_endline (List.hd (dyp.Dyp.next_lexeme ())) }
  | "efg" { () }
```

Calling the lexer on `a/*efg*/bcd`, `next_lexeme` returns `["efg";"/*"]`, whereas the lexer reads `/*efg*/`, skips it and then reads the lexeme `bcd`.

These cases may not be typical but you should watch out for them if you intend to use `next_lexeme`.

6 Dynamic extension of the grammar

6.1 Adding rules

Dynamic changes to the grammar are performed by action code. To add rules to the grammar from an action code one states the character '`@`' before the code of the action and uses the parser commands list (a list of values of type `dyp_action`, see section 7.2 for more information about it). The constructor that is used for this purpose is:

```
  | Add_rules of
      (rule * (('token,'obj,'gd,'ld,'l) dypgen_toolbox ->
      ('obj list -> 'obj * ('token,'obj,'gd,'ld,'l) dyp_action list))) list
```

Where `dypgen_toolbox` is the type of the record `dyp` (see section 7.1), and the type `obj` is explained a bit further. To add several rules and their respective actions to the grammar, the action returns the list of the corresponding couples `(rule, action)` as argument to the constructor `Add_rules` in the list of parser commands. For instance the action may look like:

```
@{ returned_value, [Add_rules [(rule1,action1);(rule2,action2);(rule3,action3)]] }
```

The types used to construct rules are:

```
  type 'a nt_prio =
    | No_priority
    | Eq_priority of 'a
    | Less_priority of 'a
    | Lesseq_priority of 'a
    | Greater_priority of 'a
    | Greatereq_priority of 'a

  type regexp =
    | RE_Char of char
    | RE_Char_set of (char * char) list
    | RE_Char_set_exclu of (char * char) list
```

```
    | RE_String of string
    | RE_Alt of regexp list
    | RE_Seq of regexp list
    | RE_Star of regexp
    | RE_Plus of regexp
    | RE_Option of regexp
    | RE_Name of string (* name of a regexp declared with let *)
    | RE_Eof_char

  type symb =
    | Ter        of string
    | Ter_NL     of string
    | Non_ter    of string * (string nt_prio)
    | Non_ter_NL of string * (string nt_prio)
    | Regexp     of regexp
    | Regexp_NL  of regexp

  type rule_options = No_layout_inside | No_layout_follows

  type rule = string * (symb list) * string * (rule_options list)
```

In the type `rule`, the first string is the non terminal of the left-hand side of the rule, `symb list` is the list of symbols in the right-hand side of the rule, the second string is the name of the priority that is returned by this rule. If you don't want your rule to return any particular priority then use `"default_priority"`. The `rule_options list` tells whether layout characters are allowed to be matched for this rule inside and after the rule. `No_layout_inside` is equivalent to the use of the character '`!`' at the beginning of the right-hand side in the parser definition and `No_layout_follows` is equivalent to '`-`' at the end of the right-hand side. The suffix '`_NL`' in the constructors of the type `symb` tells the parser that the symbol must not be preceded by layout characters, it amounts to the use of '`-`' before a symbol in a right-hand side in the parser definition (see section 5.7 for more information). These types are part of the module `Dyp` which is not open by default.

For example the rule `expr: LPAREN expr RPAREN` is written:

```
let rule_paren =
  ("expr",
  [Ter "LPAREN"; Non_ter ("expr",No_priority); Ter "RPAREN"],
  "default_priority",[])
```

The rule `expr: "(" expr ")"` is written:

```
let rule_paren =
  ("expr",
  [Regexp (RE_String "("); Non_ter ("expr",No_priority); Regexp (RE_String ")")],
  "default_priority",[])
```

The rule `expr: expr(<=pp) "+" expr(<pp) pp` is written:

```
let rule_plus =
  ("expr",
    [Non_ter ("expr",Lesseq_priority "pp");
     Regexp (RE_String "+");
     Non_ter ("expr",Less_priority "pp")],
  "default_priority",[])
```

The type of actions is:

```
('token,'obj,'gd,'ld,'l) dypgen_toolbox ->
'obj list ->
'obj * ('token,'obj,'gd,'ld,'l) dyp_action list
```

For example the action bound to the rule

```
expr: expr(<=pp) "+" expr(<pp) pp
```

can be written:

```
let action_plus _ l =
  let x1, x2 = match l with
    | [Obj_expr x1;_;Obj_expr x2] -> x1, x2
    | _ -> failwith "action_plus"
  in
  (Obj_expr (x1+x2)), []
```

The constructor `Obj_expr` is part of the type `obj` which is discussed in the following subsection.

6.2 The type `obj`

The type `obj` is a sum of constructors with each constructor corresponding to a terminal or a non terminal. For each symbol of name `symbol` in the grammar, the corresponding constructor of the type obj is `Obj_symbol`. `Obj_symbol` has an argument if `symbol` is a non terminal or a token with argument. The type of this argument is the type of the argument of the corresponding token or the type of the value returned by the corresponding non terminal.

When you write the function that builds the action associated with a rule added dynamically to the grammar, you have to use one of these constructors for the result of the action. If the constructor is not the good one, i.e. the one that is associated with the non terminal in the left-hand side of the rule, then an exception will be raised when reducing by this rule. This exception is:

```
exception Bad_constructor of (string * string * string)
```

where the first string represents the rule, the second string represents the constructor that was expected for this left-hand side non terminal and the third string is the name of the constructor that has been used.

If you want several symbols to have the same constructor then you can use the directive:

```
%constructor Cons %for symb1 symb2 ...
```

where `Cons` is the name of the common constructor you want to assign for all these symbols, and `symb1`, `symb2` can be either a non terminal (an identifier beginning with a lower case letter) or a terminal (an identifier beginning with an upper case letter). Of course all these symbols need to return values of the same type, in particular you can share the same constructor between different non terminals and tokens associated with one argument (all of them of the same type), but you cannot share a constructor between tokens without argument and tokens with arguments or non terminals. Regardless of whether you use polymorphic variants (see below) or not, you should not write `‘Cons` here but just `Cons`. You can also use the keyword `%constructor` to declare constructors that are not used by any token or non terminal in the initial grammar but that may be used for new non terminals when new rules are added dynamically (see section 6.5). For instance:

```
%constructor Cons1 Cons2 Cons3
```

Note that `obj` is actually a polymorphic type constructor. The types of the arguments of the constructors that accept an argument that are not known to dypgen are its type parameters. But these type parameters are not explicitly written in this manual to avoid clutter. See the next subsection for an example of type constructor `obj`. Note that if your grammar has a lot of symbols, the maximal number of non-constant constructors (246) may be reached. Then use the option `--pv-obj` with `dypgen`. With this option, dypgen uses polymorphic variants instead of constructors. Another way to avoid reaching the maximum number of constructors is to share constructors between several symbols, for instance to use a constructor `Void` for all the tokens that have no argument. When using polymorphic variants sharing constructor may make the compilation of the generated code less demanding because it uses fewer polymorphic variants.

In addition to the constructors assigned to terminals and non terminals symbols, the type `obj` owns the following constructors:

```
Lexeme_matched of string
```

It is the constructor for the strings returned by regular expressions. This constructor is present even if you don't use dypgen lexer generator. When dypgen is used as the lexer generator there is also one constructor

```
Lex_lexer_name
```

for each auxiliary lexer (where `lexer_name` is the name of the lexer), and one constructor

```
Lex_lexer_name_Arg_arg_name
```

for each parameter of the lexer (where `arg_name` is the name of the parameter). The user won't have to deal with these constructors but you should avoid introducing constructor that begins with `Lex_` or giving a name that contains `_Arg_` to a lexer.

The `.mli` file generated by dypgen includes the declaration of the type `obj` (when neither `--pv-obj` nor `--no-obj` is used).

6.3 Example

```
{ open Dyp

let rule_plus = ("expr", [
  Non_ter ("expr",No_priority);
  Regexp (RE_String "+");
  Non_ter ("expr",No_priority)
  ], "default_priority", true)
(* define the rule: expr: expr "+" expr *)

let action_plus = (fun dyp l ->
  let x1,x2 = match l with
    | [Obj_expr x1;_;Obj_expr x2] -> x1,x2
    | _ -> failwith "action_plus"
  in
  Obj_expr (x1+x2), [Dont_shift]) }

%start main

%lexer
main lexer =
' ' ->
['0'-'9']+ -> INT { int_of_string (Dyp.lexeme lexbuf) }

%parser

main: expr "\n"    { $1 }

expr:
  | INT           { $1 }
  | a expr        { $2 }

a: "&" @{ (), [Add_rules [(rule_plus, action_plus)]] }
```

Now if we parse the following input

```
& 1 + 2 + 3 + 4
```

the parser reduces `&` to `a`, the action code of this reduction adds the rule `expr: expr "+" expr` to the initial grammar which does not contain it. And the action code associated with this rule is `action_plus` which is the addition. Then the rest of the input is parsed with this new rule and the whole string is reduced to the integer 10.

The type constructor `obj` discussed in the previous subsection is:

```
type ('a, 'b, 'c, 'd) obj =
    Lexeme_matched of string
  | Obj_INT of 'a
```

```
  | Obj___dypgen_layout
  | Obj_a of 'b
  | Obj_expr of 'c
  | Obj_main of 'd
  | Dypgen__dummy_obj_cons
```

Note that with an early action we can use alternatively the following grammar:

```
main: expr "\n"    { $1 }
expr:
  | INT            { $1 }
  | "&" ...@{ (), [Add_rules [(rule_plus, action_plus)]] } expr { $3 }
```

Now the type constructor `obj` becomes:

```
type ('a, 'b, 'c, 'd) obj =
    Lexeme_matched of string
  | Obj_INT of 'a
  | Obj___dypgen_layout
  | Obj_dypgen__nt_0 of 'b
  | Obj_expr of 'c
  | Obj_main of 'd
  | Dypgen__dummy_obj_cons
```

6.4 Scope of the changes

The changes to the grammar introduced by an action do not apply anywhere in the input. When an action code changes the grammar a reduction occurs and yield a non terminal (the non terminal `a` in the previous example). Once this non terminal is itself reduced, the changes to the grammar are forgotten and the grammar which was used before the changes is used again. The scope of the changes to the grammar is the same as the scope of `local_data`.

We add parentheses to the previous example:

```
{
  (* same as previous example *)
}

%start main
%lexer /* same as previous example */

%parser
main: expr "\n"    { $1 }

expr:
  | INT            { $1 }
  | a expr         { $2 }
  | "(" expr ")" { $2 }

a: "&" @{ (), [Add_rules [(rule_plus, action_plus)]] }
```

The input

```
(& 1 + 2 + 3 + 4)
```

is correct, but

```
(& 1 + 2) + 3 + 4
```

is not and raises `Dyp.Syntax_error`. In order to reduce `(& 1 + 2)` the parser reduces `& 1 + 2` to `a expr` and then to `expr`. At this moment the old grammar applies again and the rule `expr: expr "+" expr` does not apply any more. This is why the parser cannot shift `+` after `(& 1 + 2)`. In the directory `demos/sharp` there is an example which is similar to this one.

It is possible to extend the scope of the grammatical changes, but you have to tell it to the parser. The constructor

```
Keep_grammar
```

of the type `dyp_action` is available to do that. If `Dyp.Keep_grammar` is returned in the list of instructions to the parser, then the grammar that was in use when the reduction of the last non terminal of the right-hand side of the current rule occured is kept instead of being forgotten. Using:

```
@{ returned_value, [Dyp.Keep_grammar] }
```

extends the scope of the grammar the same way that

```
@{ returned_value, [Dyp.Local_data dyp.last_local_data] }
```

extends the scope of `local_data` (see section 4.5).

Extensions of the scope of the changes to the grammar are useful when you want to use a left-recursive rule instead of a right-recursive one. Left-recursive rules are more efficient than right-recursive ones with a GLR parser driven by an LR(0) automaton, which is currently what dypgen generates. For an example of use of `Keep_grammar` with a left-recursive rule, see the example `tinyML`.

Using `Dyp.Keep_grammar` is expensive since it needs to rebuild an automaton. There is an exception, when the rule is left-recursive then no additional construction of an automaton is needed.

Note that `Dyp.Keep_grammar` has no effect when the current action itself changes the grammar.

6.5 Adding new non terminals

To add a new non terminal to the grammar just use its string in some new rules and bind this new non terminal to an existing constructor of the type `obj`. To do this binding place the following in the parser commands list:

```
Bind_to_cons [(nt,cons)]
```

is used to insert a new non terminal which string is `nt` and which associated constructor has the string `cons`. The string `nt` can be any string, the string `cons` must be either:

- A valid constructor of the form `Obj_nt` where `nt` is a non terminal which has not been assigned a different constructor than its default.
- Or a constructor that has been declared with the keyword `%constructor` and without using the backquote even if the option `--pv-obj` is used.

If the non terminal which string is `nt` is already bound to a constructor then there are two possible outcomes:

- The string `cons` is the same as the one that this non terminal is already bound to, then nothing is changed.
- The string `cons` is different from the one that this non terminal is already bound to, then the parser raises:

  ```
  exception Constructor_mismatch of (string * string)
  ```

  where the first string is the one representing the constructor that the non terminal was previously bound to and the second string is `cons`.

Note that if an action returns `Keep_grammar` and uses `Bind_to_cons binding_list` in the parser commands list then the bindings introduced by `Bind_to_cons` will not be taken into account.

Note that you cannot define new terminal symbols. If you add a new rule with a token that is not in the initial grammar then the parser will raise the exception:

```
exception Undefined_ter of string
```

The string is the name of the undefined token. If you want to extend the lexer then use dypgen as the lexer generator and use regular expressions in the right hand-side of the new rules that need the lexer to be extended.

Note that you can also include non terminals in the initial grammar without being part of any rule. To do this use the keyword `%non_terminal` followed by the names of the non terminals you want to include, like:

```
%non_terminal nt1 nt2 nt3
```

This directive is put in the section after the header, before the grammar.

For a complete example of grammar extension, see appendix C. It describes a small language that is somewhat extensible.

6.6 Extending dynamically priority data

To make the relation true between priorities, use the constructor `Relation` of the parser commands list:

```
Relation of string list list
```

For instance the following parser commands list

```
[ Relation [["p1";"p2"];["p3";"p4";"p5"]] ]
```

adds the relations: `p1<p2` and `p3<p4<p5` to the priority data. (where `p3<p4<p5` is transitively closed, i.e. `p3<p5` does also hold)

The scope of the changes to the priority data is the same as the scope of the changes to the grammar. In particular the parser command `Keep_grammar` extends this scope the same way it does for the scope of new rules.

Note that it is not possible to make the relation false between two priorities for which the relation is true at parsing time. You can see the priorities as the vertices of a graph and the relation as directed edges between these vertices. You can dynamically add new vertices and edges, but not remove them. Of course when the parsing goes out of the scope of the changes, the priority data returns to its previous state.

To add a new priority to the grammar just use it in a rule or with `Relation`.

You can find a very basic example of extension of priority data in the directory `calc_ext`. This is the example of a calculator that only has the plus operator and where the times operator can be added with the use of the character `&` in the string. Initially the priorities are `pi<pp`, when the grammar is extended to include the times operator, the priority data is changed to `pi<pt<pp`.

7 Parsing control structures

7.1 The record `dyp` and the type `dypgen_toolbox`

The record `dyp` is accessible in any action code, its type is `dypgen_toolbox`, it is defined in the module `Dyp`:

```
type ('token,'obj,'gd,'ld,'lexbuf) dypgen_toolbox = {
  parser_pilot : ('token,'obj,'gd,'ld,'lexbuf) parser_pilot;
  global_data : 'gd;
  local_data : 'ld;
  last_local_data : 'ld;
  next_lexeme : unit -> string list;
  symbol_start : unit -> int;
  symbol_start_pos : unit -> Lexing.position;
  symbol_end : unit -> int;
  symbol_end_pos : unit -> Lexing.position;
  rhs_start : int -> int;
  rhs_start_pos : int -> Lexing.position;
  rhs_end : int -> int;
  rhs_end_pos : int -> Lexing.position;
  print_state : out_channel -> unit;
  print_grammar : out_channel -> unit;
}
```

Where `'gd` and `'ld` are replaced by the type for global data and local data chosen by the user (and infered by Caml), and `'obj` is replaced by the type `obj` discussed in section 6.2, and `'lexbuf` is replaced by the type of the lexer buffer. To know more about the field `parser_pilot` see section 7.3, and for more information about `next_lexeme` see section 5.8.

7.2 The parser commands list and the type `dyp_action`

To give instructions to the parser from the user actions one returns a list of values of type `dyp_action`, called the parser commands list, along with the returned AST. This list is returned by the action when the code is preceded by the character '`@`' and it is returned by the actions introduced at parsing time.

```
type ('token,'obj,'gd,'ld,'lexbuf) dyp_action =
  | Add_rules of
      (rule * (('token,'obj,'gd,'ld,'lexbuf) dypgen_toolbox ->
      ('obj list -> 'obj * ('token,'obj,'gd,'ld,'lexbuf) dyp_action list))) list
  | Bind_to_cons of (string * string) list
  | Dont_shift
  | Global_data of 'gd
  | Keep_grammar
  | Local_data of 'ld
  | Next_grammar of out_channel
  | Next_state of out_channel
  | Parser of ('token,'obj,'gd,'ld,'lexbuf) parser_pilot
  | Relation of string list list
```

Here are the sections where each of these constructors are discussed:

Constructor	See section	Constructor	See section	Constructor	See section
`Add_rules`	6.1	`Keep_grammar`	6.4	`Parser`	7.4
`Bind_to_cons`	6.5	`Local_data`	4	`Relation`	6.6
`Dont_shift`	3.4	`Next_grammar`	9.3		
`Global_data`	4	`Next_state`	9.3		

The type `dyp_action` is also used with the function `update_pp`, see section 7.3.

Because of the recursive nature of the types `dyp_action` and `dypgen_toolbox` the typechecker of ocamlc complains in some cases, like when using an early action that returns a parser commands list (i.e. beginning with `...@{`). The error message looks like:

```
This expression has type
  ('a * unit, 'b) obj * ('c, ('d, int) obj, 'e, 'f, 'g) Dyp.dyp_action list
but is here used with type 'a * ('h, 'a, 'i, 'j, 'k) Dyp.dyp_action list
```

In such cases you will have to use the option `-rectypes` of ocamlc and use `--ocamlc "-rectypes"` with dypgen or the option `command` (These options are documented in 9.7)

7.3 parser_pilot, parse, lexparse and update_pp

The record `dyp` available inside the action code has a field:

```
parser_pilot : ('token,'obj,'global_data,'local_data,'lexbuf) Dyp.parser_pilot;
```

that describes the current parser with information such as the current grammar, its corresponding parse table and the current global and local data. The type is

```
type ('token,'obj,'global_data,'local_data,'lexbuf) parser_pilot = {
  pp_dev : ('token,'obj,'global_data,'local_data,'lexbuf) parsing_device;
  pp_par : ('token,'obj,'global_data,'local_data,'lexbuf) parser_parameters;
  pp_gd : 'global_data;
  pp_ld : 'local_data }
```

The `.ml` file generated by dypgen defines the value:

```
val pp : unit -> ('token,'obj,'global_data,'local_data,'lexbuf) Dyp.parser_pilot
```

which is declared in the `.mli` generated file. It is defined after the grammar, therefore it is not available in the header and in the grammar. This value describes the parser defined in the `.dyp` file. The global and local data included inside are the values `global_data` and `local_data` defined by the user (or the default otherwise).

The following function is defined in the module `Dyp` of the library and makes possible to parse for any non terminal symbol of the grammar when you use dypgen as the lexer generator.

```
val lexparse :
  ('token,'obj,'global_data,'local_data,'obj dyplexbuf) parser_pilot ->
  string ->
  ?global_data:'global_data ->
  ?local_data:'local_data ->
  ?match_len:[`longest|`shortest] ->
  ?choose_token:[`first|`all] ->
  'obj dyplexbuf ->
  (('obj * string) list)
```

If you use an external lexer generator like ocamllex then the corresponding function is:

```
val parse :
  ('token,'obj,'global_data,'local_data,'lexbuf) parser_pilot ->
  string ->
  ?global_data:'global_data ->
  ?local_data:'local_data ->
  ?match_len:[`longest|`shortest] ->
  ?lexpos:('lexbuf -> (Lexing.position * Lexing.position)) ->
  ('lexbuf -> 'token) ->
  'lexbuf ->
  (('obj * string) list)
```

For instance the following expression inside a user action:

```
parse dyp.parser_pilot "expr" lexfun lexbuf
```

parses with the current grammar and the current global and local data with the lexer `lexfun` and the lexer buffer `lexbuf` and tries to reduce to the non terminal `expr`. If you want to have access to your lexer inside the actions, you may put it in `global_data` or `local_data`. Assuming you want to put it in `local_data`, then when you define `local_data` in the header you may put a dummy lexer of the right type:

```
let local_data = fun _ -> LPAREN
```

where `LPAREN` is any token. And when you call the parser, use the label `~local_data:` to have `local_data` contains your real lexer.

Assuming that a parser is defined in a file named `my_parser.dyp` then the following expression in a file different from the parser:

```
parse (My_parser.pp ()) "program" lexfun lexbuf
```

parses with the grammar defined in `my_parser.dyp` and the values `global_data` and `local_data` defined by the user. It tries to reduce to the non terminal `program`.

The non terminal passed as second argument to `parse` does not need to be declared as a start non terminal in the `.dyp` file. Any non terminal of the grammar can be used. The default value for `match_len` is `` `longest `` (see section 9.8 for more information about `` `shortest `` and `` `longest ``). The default value for `choose_token` is `` `first `` (see the end of the section 2.1 for more information about this argument).

The following function is defined in the module `Dyp` of the library and makes possible to modify a value of type `parser_pilot`.

```
val update_pp :
  ('token,'obj,'global_data,'local_data,'lexbuf) parser_pilot ->
  ('token,'obj,'global_data,'local_data,'lexbuf) dyp_action list ->
  ('token,'obj,'global_data,'local_data,'lexbuf) parser_pilot
```

Note: when you use this function, only the following constructors of the type `dyp_action` have an effect on the parser:

```
Add_rules
Bind_to_cons
Global_data
Local_data
Relation
```

To know more about the type `dyp_action` see section 7.2.

7.4 Saving and loading a parser

It is possible to save to a file and load from a file a parser. This may be useful to save the time of generation of the automaton for an extended grammar. To do this you need to use the field `parser_pilot.pp_dev` of the record `dyp` and the constructor

```
Parser of ('token,'obj,'global_data,'local_data,'lexbuf) parsing_device
```

of the parser commands list. The field `parser_pilot.pp_dev` gives you access to the current parser and make you able to save it to a file using the `Marshal` module of the Caml standard library (you have to use the flag `Closures` since a value of type `parsing_device` contains functional values). To load a parser and use it, you need to return it to the current parser with the constructor `Parser` in a parser commands list.
For instance the demo program `tinyML` could contain the following productions to save and load the `parsing_device`.

```
statements:
  | { [] }
  | statements statement       @{ $2::$1, [Keep_grammar] }
  | statements load_statement  @{ $2@$1, [Keep_grammar] }
  | statements "save" STRING<filename> ";" ";" @{
      let oc = open_out filename in
      Marshal.to_channel oc ($1, dyp.parser_pilot.pp_dev) [Marshal.Closures];
      close_out oc;
      $1, [Keep_grammar] }

load_statement: "load" STRING<filename> ";" ";"
  @{let ic = open_in filename in
    let stmt_list, pdev = Marshal.from_channel ic in
    close_in ic;
    stmt_list, [Parser pdev] }
```

The files `test_list_syntax.tiny` and `test_load_parser.tiny` test this feature. The file `test_list_syntax.tiny` is

```
define
  list_contents := expr(x) = List(x,Nil)
  and list_contents := expr(x) ";" list_contents(y) = List(x,y)
  and expr := "[" "]" = Nil
  and expr := "[" list_contents(x) "]" = x
  and expr := expr(x) "::" expr(y) = List(x,y)
;;

let rec append arg = match arg with
  | ([],list) -> list
  | ((head::tail),list) -> (head::(append (tail,list)))
;;

define
  expr := expr(x) "@" expr(y) = append (x,y)
;;

save "list_syntax";;
```

When interpreted it creates a file `list_syntax` containing a parser for the syntax of tinyML and an extension parsing the syntax of lists.
The file `test_load_parser.tiny` is

```
load "list_syntax";;

let rec reverse l = match l with
  | [] -> []
  | head::tail -> ((reverse tail)@[head])
;;

reverse [0;1;2;3];;
```

When interpreted it outputs

```
= List(3,List(2,List(1,List(0,Nil))))
```

It is possible to save the automaton without functional values and to attach them to the `parsing_device` after loading it. To do this, two functions are available:

```
val function_free_pdev :
  ('t,'o,'gd,'ld,'lb) parsing_device -> ('t,'o,'gd,'ld,'lb) parsing_device

val import_functions :
  ('t,'o,'gd,'ld,'l) parsing_device ->
  ('t,'o,'gd,'ld,'l) parser_pilot ->
  (rule * (('t,'o,'gd,'ld,'l) dypgen_toolbox ->
      'o list -> 'o * ('t,'o,'gd,'ld,'l) dyp_action list)) list ->
  ('t,'o,'gd,'ld,'l) parsing_device
```

The function `function_free_pdev` returns a `parsing_device` without any functional value. And `import_functions` allows to attach them. For instance the demo program `tinyML` contains the following productions:

```
statements:
  | { [] }
  | statements statement
      @{ $2::$1, [Keep_grammar] }
  | statements load_statement
      @{ $2@$1, [Keep_grammar] }
  | statements "save" STRING<filename> ";" ";" @{
      let oc = open_out filename in
      Marshal.to_channel oc
        ($1, (snd dyp.global_data), function_free_pdev dyp.parser_pilot.pp_dev)
        [];
      close_out oc;
      $1, [Keep_grammar] }

statement:
```

```
/* ... */
  | "define" define_cont ";" ";" @{
      let bind_cons = List.map (fun (s,_,_) -> (s,"EXPR")) $2 in
      Define_syntax,
      let gd = (fst dyp.global_data), $2@(snd dyp.global_data) in
      [Global_data gd;
      Add_rules (List.map (a_define_in dyp) $2); Bind_to_cons bind_cons] }

load_statement: "load" STRING<filename> ";" ";" @{
      let ic = open_in filename in
      let stmt_list, define_cont, pdev = Marshal.from_channel ic in
      close_in ic;
      let ral = List.map (a_define_in dyp) define_cont in
      let pdev = import_functions pdev dyp.parser_pilot ral in
      stmt_list, [Parser pdev] }
```

Here the information about all the grammar rules that have been added to the initial grammar and the information to build the corresponding user actions are stored in `global_data` and marshalled too when saving the `parsing_device`.

8 Names defined by dypgen

To avoid names conflicts you should not use identifier beginning with `__dypgen_` and take into account the names that are listed in this section. You sould not give your non terminal symbols names beginning with `dypgen__`.

8.1 Types and values defined in the `.ml` file

The following values are available in the code of the parser. They are defined before the header but after the code introduced by `%mltop`.

```
type token =
  | TOKEN_1 of t1
  | TOKEN_2 of t2
...
```

The type `token` is defined only if an external lexer generator is used and the options `--pv-token` and `--noemit-token-type` are not used.

```
type ('nt1,'nt2,...) obj =
  | Obj_TOKEN_1 of t1
  | Obj_TOKEN_2 of t2
...
  | Obj_non_ter_1 of 'nt1
  | Obj_non_ter_2 of 'nt2
...
```

The type `obj` is defined only if the options `--pv-obj` and `--no-obj` are not used.

```
val global_data : int ref (* by default, can be defined by the user *)
val local_data : int ref (* by default, can be defined by the user *)
val global_data_equal : 'a -> 'a -> bool (* is (==) by default *)
val local_data_equal : 'a -> 'a -> bool (* is (==) by default *)

val dyp_merge : 'a list -> 'a -> 'a list
val dyp_merge_nt1 : 'a list -> 'a -> 'a list
val dyp_merge_nt2 : 'a list -> 'a -> 'a list
...

val dypgen_lexbuf_position : Lexing.position * Lexing.position
val dypgen_match_length : [ `shortest | `longest ]
val dypgen_choose_token : [ `first | `all ]

module Dyp_symbols:
sig
  val get_token_name : token -> int
  val str_token : token -> string
  val ter_string_list : (string * token_name) list
end

module Dyp_priority_data:
sig
  val relations : string list list
end
```

In addition the following module names are used:

```
module Dyp_symbols_array
module Dyp_aux_functions
```

8.2 Types and values defined in the `.mli` file

The `.mli` file generated by dypgen declares automatically the types `token` when using an external lexer generator and `obj` (if `--pv-token` or `--noemit-token-type` is used then `token` is not declared, if `--pv-obj` or `--no-obj` is used then `obj` is not declared, see 9.6 for information about `--pv-obj`). It then declares the value `pp` (see section 7.3 for more information about it) and one function for each entry point of the grammar declared as such with the keyword `%start`.

The types written by dypgen are extracted as strings from a file ending with `.extract_type` created by calling `ocamlc -i`. As `ocamlc -i` does not output a compilable file, dypgen does the following modifications to the strings. Polymorphic variants are fixed: `_[>`, `_[<`, `[>` and `[<` are replaced with `[`. Type variables beginning with `'_` (like `'_a`) are replaced with `unit`. These automatic modifications may be the cause of puzzling type errors in some cases. Avoid such errors with type annotations.

You may want to use some options with ocamlc when dypgen calls `ocamlc -i`, like the paths where ocamlc must look for modules. Then uses the options `--ocamlc` or `--command` documented in section 9.7.

8.3 Adding code to the `.mli` file and at the top of the `.ml` file

The general frame of a `.dyp` file is:

```
%mltop { (* optional OCaml code *) }

{ (* optional OCaml code: "the header" *) }

/* parser informations introduced with keywords (priorities, tokens, ...) */

%%

/* the grammar */

{ (* optional OCaml code: "the trailer" *) }

%mlitop { (* optional OCaml interface code *) }
%mlimid { (* optional OCaml interface code *) }
%mli { (* optional OCaml interface code *) }
```

The keyword `%mltop { }` makes possible to add code to the top of the `.ml` generated file. The code inside the braces is copied to the beginning of the `.ml` file, before the values defined by dypgen. The keyword `%mltop` is used first in the parser definition, before the optional header.

The keyword `%mlitop { }` makes possible to add code to the top of the interface file of the parser. The code inside the braces is copied to the beginning of the `.mli` file.

With the keyword `%mlimid { }` the code inside the braces is copied between the declaration of type token and the declaration of the entry point functions in the `.mli` file.

The keyword `%mli { }` makes possible to add code at the end of the interface file of the parser. The code inside the braces is appended to the end of the `.mli` file. The keyword `%mli` is used at the end of the parser definition.

Remark: the block `%mlitop { }` and the trailer make possible to encapsulate the parser definition in a functor.

8.4 No generation of the `.mli` file and other options

With the option `--no-mli` dypgen does not generate a `.mli` file.

The option `--noemit-token-type` prevents dypgen from writing the type `token` in the generated code. This is useful when you want to define this type in another file.

The option `--no-pp` prevents dypgen from declaring pp in the `.mli` file.

The option `--no-obj` prevents dypgen from declaring the type `obj` in the `.mli` file (it has no effect if `pp` is declared).

8.5 Defining the token type in a separate file

To do this, use the option `--noemit-token-type` when calling dypgen, and do the following: Supposing your parser file is `parser.dyp` and your token type is the type `t` defined in the file `token.ml`, put this at the beginning of `parser.dyp`:

```
%mltop{
open Token
type token = Token.t
}
```

and put this at the end of `parser.dyp`:

```
%mlitop{
type token = Token.t
}
```

8.6 The module `Dyp` of the library

Names defined in the module `Dyp` are listed here:

```
val dypgen_verbose : int ref

type 'a nt_prio =
  | No_priority
  | Eq_priority of 'a
  | Less_priority of 'a
  | Lesseq_priority of 'a
  | Greater_priority of 'a
  | Greatereq_priority of 'a

type regexp =
  | RE_Char of char
  | RE_Char_set of (char * char) list
  | RE_Char_set_exclu of (char * char) list
  | RE_String of string
  | RE_Alt of regexp list
  | RE_Seq of regexp list
  | RE_Star of regexp
  | RE_Plus of regexp
  | RE_Option of regexp
  | RE_Name of string (* name of a regexp declared with let *)
  | RE_Eof_char
```

```
type symb =
  | Ter of string
  | Non_ter of string * (string nt_prio)
  | Regexp of regexp
  | Ter_NL of string
  | Non_ter_NL of string * (string nt_prio)
  | Regexp_NL of regexp

type rule_options = No_layout_inside | No_layout_follows
type rule = string * (symb list) * string * rule_options list
type nt_cons_map

type ('obj,'gd,'ld) merge_function =
  ('obj * 'gd * 'ld) list -> ('obj list * 'gd * 'ld)

type debug_infos = {
  prt_state : out_channel -> unit;
  prt_grammar : out_channel -> unit; }

type ('t,'o,'gd,'ld,'l) action =
    Dypgen_action of (...)
and ('token,'obj,'global_data,'local_data,'lexbuf) parser_pilot

type ('token,'obj,'gd,'ld,'l) dypgen_toolbox = {
  parser_pilot : ('token,'obj,'gd,'ld,'l) parser_pilot;
  global_data : 'gd;
  local_data : 'ld;
  last_local_data : 'ld;
  next_lexeme : unit -> string list;
  symbol_start : unit -> int;
  symbol_start_pos : unit -> Lexing.position;
  symbol_end : unit -> int;
  symbol_end_pos : unit -> Lexing.position;
  rhs_start : int -> int;
  rhs_start_pos : int -> Lexing.position;
  rhs_end : int -> int;
  rhs_end_pos : int -> Lexing.position;
  print_state : out_channel -> unit;
  print_grammar : out_channel -> unit;
}

type ('token,'obj,'gd,'ld,'l) dyp_action =
  | Add_rules of
      (rule * (('token,'obj,'gd,'ld,'l) dypgen_toolbox ->
      ('obj list -> 'obj * ('token,'obj,'gd,'ld,'l) dyp_action list))) list
  | Bind_to_cons of (string * string) list
```

```
  | Global_data of 'gd
  | Keep_grammar
  | Local_data of 'ld
  | Next_grammar of out_channel
  | Next_state of out_channel
  | Relation of string list list
  | Dont_shift

exception Giveup
exception Undefined_nt of string
exception Undefined_ter of string
exception Bad_constructor of (string * string * string)
exception Constructor_mismatch of (string * string)
exception Syntax_error

val keep_all : ('obj,'gd,'ld) merge_function
val keep_one : ('obj,'gd,'ld) merge_function
val dummy_lexbuf_position : 'a -> (Lexing.position * Lexing.position)

module Tools

val print_regexp : regexp -> string

type 'obj dyplexbuf

val lexeme : 'obj dyplexbuf -> string
val lexeme_char : 'obj dyplexbuf -> int -> char
val lexeme_start : 'obj dyplexbuf -> int
val lexeme_end : 'obj dyplexbuf -> int
val lexeme_start_p : 'obj dyplexbuf -> Lexing.position
val lexeme_end_p : 'obj dyplexbuf -> Lexing.position
val flush_input : 'obj dyplexbuf -> unit

val from_string :
  ('token,'obj,'global_data,'local_data,'lexbuf) parser_pilot ->
  string ->
  'obj dyplexbuf

val from_channel :
  ('token,'obj,'global_data,'local_data,'lexbuf) parser_pilot ->
  in_channel ->
  'obj dyplexbuf

val from_function :
  ('token,'obj,'global_data,'local_data,'lexbuf) parser_pilot ->
  (string -> int -> int) ->
  'obj dyplexbuf
```

```
val dyplex_lexbuf_position :
  'obj dyplexbuf -> (Lexing.position * Lexing.position)

val std_lexbuf : 'obj dyplexbuf -> Lexing.lexbuf
val set_newline : 'obj dyplexbuf -> unit
val set_fname : 'obj dyplexbuf -> string -> unit

val make_parser

val update_pp :
  ('token,'obj,'global_data,'local_data,'lexbuf) parser_pilot ->
  ('token,'obj,'global_data,'local_data,'lexbuf) dyp_action list ->
  ('token,'obj,'global_data,'local_data,'lexbuf) parser_pilot

val lex :
  string -> (* name of the auxiliary lexer that is called *)
  'obj list ->
  'obj dyplexbuf ->
  'obj

val parse :
  ('token, 'obj,'global_data,'local_data,'lexbuf) parser_pilot ->
  string ->
  ?global_data:'global_data ->
  ?local_data:'local_data ->
  ?match_len:[`longest|`shortest] ->
  ?lexpos:('lexbuf -> (Lexing.position * Lexing.position)) ->
  ('lexbuf -> 'token) ->
  'lexbuf ->
  (('obj * string) list)

val lexparse :
  ('token, 'obj,'global_data,'local_data,'obj dyplexbuf) parser_pilot ->
  string ->
  ?global_data:'global_data ->
  ?local_data:'local_data ->
  ?match_len:[`longest|`shortest] ->
  ?choose_token:[`first|`all] ->
  'obj dyplexbuf ->
  (('obj * string) list)
```

9 Other features

9.1 Preprocessing with cpp

dypgen is compatible with the C preprocessor. The option `--cpp` makes dypgen call cpp on the input file before processing it.

This allows to have the grammar definition spread over several files and to have some form of parameterization for the rules via the macros.

Here is an example with the calculator. The file `calc_parser.dyp`:

```
%start main

%relation pi<pt<pp
%layout [' ' '\t']

%parser

main: expr "\n" { $1 }

#include "expr.dyp"
```

The file `expr.dyp`:

```
#define INFIX(op,p) expr(<=p) #op expr(<p) { $1 op $3 } p

expr:
  | ['0'-'9']+     { int_of_string $1 }  pi
  | "-" expr(=pi)            { -$2 }     pi
  | "(" expr ")"             { $2 }      pi
  | INFIX(+,pp)
  | INFIX(-,pp)
  | INFIX(*,pt)
  | INFIX(/,pt)
```

The option `--cpp-options "options"` allows to make dypgen pass some options to cpp. It is useful to pass the flag `-w` to cpp to avoid the warning messages. `--cpp` is not needed when using `--cpp-options`.

9.2 Generated documentation of the grammar

A quick and ugly perl script to generate a BNF documentation for your grammar is provided with dypgen. This script is not guaranteed to generate anything of interest but it works as a proof of concept. The grammar generated in this way is more readable than the original dypgen file, and since a lot of conflict resolving can be done inside the actions, the grammar is usually much more

concise than a corresponding Ocamlyacc grammar. The generated file can thus be used as such in the documentation.

You invoke this script by the following command:

```
dyp2gram.pl path_to/parser.dyp path_to/parser.txt
```

The file `parser.txt` does not contain the OCaml parts of the file `parser.dyp` (action, preamble, etc). Everything else is included without changes except for the following rules:

- The comments starting with `/*--` are removed. Other dypgen comments are kept. Remark: the OCaml comments are removed together with all OCaml code.
- If a `%token` line is annotated with a comment like `/* -> 'xxxxx' */` then, in every action, the terminal is replaced by the provided string `xxxxx` and the `%token` line is removed. This allows to replace `PLUS` by `+`, removing the need for a separate documentation for the lexer.
- If a `%token` line is annotated with a comment like `/*:= 'xxxxx' -> 'yyyyy' */` then the same as above applies, but the `%token` line is not removed and the comment is just replaced by the given definition "`:= xxxxxx`". This allows to put the definition of terminal like `IDENT`, `INT`, ... inside the generated file.
- If a `%token` line is annotated with a comment like `/*:= 'xxxxx' */`, the `%token` line is kept, but no substitution is performed.
- All other `%token` lines are kept unchanged

When a rule can parse the empty stream, it disappears because the action disappears. It is thus a good idea to put a comment like in

```
/*
 * telescopes: lists of (typed) identifiers
 */
telescope:
    /* possibly empty */
        { [] }
  | argument telescope
        { $1::$2 }
```

As an example, look at the grammar text file generated from the dypgen parser for `tinyML_ulex` (in the `demo` directory)...

9.3 Information about the parsing

The user can assign the following reference that is part of the module `Dyp`:

```
val dypgen_verbose: int ref
```

in the header of the parser definition. The value `1` makes the parser print information about dynamically built automata on the standard output. The value `2` adds the following information: number of reductions performed while parsing and the size of the graph-structured stack of the parser. Any value `>2` makes the parser logs information in a file about the parsing which are useful for debugging `dypgen` but unlikely to be useful for the user. Setting a value `>2` currently breaks re-entrancy.

The following functions may be useful for debugging purpose:

```
dyp.print_state out_channel;
```

prints the current state of the automaton on `out_channel`.

```
dyp.print_grammar out_channel;
```

prints the current grammar on `out_channel`.

```
Next_state of out_channel
```

is a constructor of the type `dyp_action` that makes possible to print the next state of the automaton (the one where the parser goes after the current reduction). Use it in your action like:

```
@{ returned_value, [Dyp.Next_state channel] }
```

```
Next_grammar of out_channel
```

is a constructor of the type `dyp_action` that makes possible to print the grammar after the current reduction (and possible changes). Use it in your action like:

```
@{ returned_value, [Dyp.Next_grammar channel] }
```

If the option `--merge-warning` is used then a warning is issued on the standard output each time a merge happens. If the `lexbuf` structure is updated by user actions in the lexer then the beginning and the end of the part of the input is given.

9.4 Warnings

Dypgen issues warnings at compile time in the following situations:

- When a non terminal symbol is only in right-hand side and not in a left-hand side and inversely (except for start symbols).
- When a lexer name contains the string '`_Arg_`'.
- When a priority is not declared.

In addition dypgen issues warning at run time when a merge is done if the option `--merge-warning` is used.

Compile time warnings become failing errors when using the command option `--Werror`.

9.5 Error

When the parser is stuck in a situation where it cannot reduce and cannot shift but had not reduced to the start symbol, it raises the exception `Dyp.Syntax_error`.

When there is in the grammar a non terminal that is in a right-hand side but never in a left-hand side (i.e. it is never defined) then the following exception is raised by the parser:

```
exception Undefined_nt of string
```

where the string is the name of this non terminal. The option `--no-undef-nt` prevents this exception from being raised.

9.6 Maximum number of constructors and using polymorphic variants

If you have a lot of tokens or non terminals then you may reach the maximum number of non-constant constructors. To solve this problem you can use the options `--pv-token` and `--pv-obj`. With `--pv-token`, the type `token` is a sum of polymorphic variants, and with `--pv-obj` the type constructor `obj` uses polymorphic variants.

9.7 Command-line options for `ocamlc` and include paths

In order to know some types, dypgen calls:

```
ocamlc -i parser.temp.ml > parser.extract_type
```

if one assume that `parser.dyp` is the input file of dypgen. If `parser.ml` needs some command-line options to be compiled, then you can pass them to `ocamlc` with the option `--ocamlc` *`string_of_options`*. For instance if you need the inlcude path `../dypgen/dyplib` and the option `-rectypes` you can call dypgen with:

```
dypgen --ocamlc "-I ../dypgen/dyplib -rectypes" parser.dyp
```

Some type errors involving the type `dyp_action` are solved by using `--ocamlc "-rectypes"` with dypgen (see section 7.2).

Alternatively, instead of the option `--ocamlc`, you can use the option `--command` which lets you state a complete command to produce the file `myparser.extract_type` (where `myparser.dyp` is the parser definition file). For instance you may use:

```
dypgen --command "ocamlfind ocamlc -package mypackage -i parser.temp.ml > \
  parser.extract_type" parser.dyp
```

And then dypgen uses the command

```
ocamlfind ocamlc -package mypackage -i parser.temp.ml > parser.extract_type
```

to generate the file `parser.extract_type`.

9.8 Longest and shortest match for the parser

The functions named after entry points of the grammar that are generated by dypgen match the shortest part of the input that can be reduced to the entry point. The user can make them match the longest by stating the following line in the header:

```
let dypgen_match_length = `longest
```

Note that by default the function `parse` of the module `Dyp` match the longest string. You can make it match the shortest using the optional argument `~match_len:`shortest`.

If you use `Dyp.keep_all` as merge function to keep all the parse trees, then you will get all the parse trees but for the same part of the input. Either the one with the shortest or the longest possible length. If you need a parser that collects several interpretations that span different lengths of the input send me an email.

Off course if your grammar makes matching the end of file mandatory then `` `shortest `` and `` `longest `` have the same effect. It makes a difference, for example, with interactive use, like in the demo program `calc`.

Note: `dypgen_match_length` has no effect on the length of the matching of the lexer, included dypgen own lexer which always selects the longest match.

9.9 Building with ocamlbuild

Here is a rule that can be used with ocamlbuild:

```
rule "dypgen"
    ~tags:["dypgen"]
    ~prods:["%.ml";]
    ~deps:["%.dyp"]
    begin fun env _ ->
     let dyp = env "%.dyp" in
       Cmd(S[A"dypgen"; A"--no-mli"; Px dyp])
    end;
```

10 Demonstration programs

The directory `demos` contains a few small programs that illustrate the use of `dypgen`.

`calc` is a simple calculator that uses priorities. It does not use dynamic change of the grammar. `calc_pattern` is the same calculator but the parser definition uses pattern matching of symbols (see section 5.2), `calc_nested` uses nested rules, `calc_ocamllex` uses ocamllex as lexer generator and `calc_ext` is an example of extending the grammar and the priority data.

`demo` is the example of appendix C, `demo_ocamllex` is (almost) the same but uses ocamllex as lexer generator.

`forest` is an example of how to use the function `dyp_merge` to yield a parse forest.

`global_data` and `local_data` are examples of using `global_data` and `local_data`. `local_data_early_action` is the same as `local_data` except that it uses an early action.

`merge_times_plus` is an example of using a merge function to enforce the precedence of the multiplication over the addition.

`position` is a small example using the functions `dyp.symbol_start`, `dyp.symbol_end`, ... which return the position of a part of the input which is reduced to a given non terminal. `position_ocamllex` uses ocamllex, `position_token_list` also uses ocamllex but it makes a list of tokens first and then parses this list.

`sharp` is a very basic demonstration of adding a rule and replacing it by another. When entering `&+` the user adds a rule which makes the character `#` like a `+` and entering `&*` makes the character `#` like a `*`.

`tinyML` is a very limited interpreted language which includes integers, tuples, constructors à la Caml, some basic pattern matching and recursive functions with one argument. It also includes a construct `define ... in` which makes possible to extend somewhat the syntax of the language by defining macros. This construct allows to add several rules at the same time as opposed to the construct `define ... in` of `demo` which can only add one rule at a time. A few input examples are included in the directory `tinyML`. To interpret them with `tinyML` do: `./tinyML test_*.tiny` where `*` is `append`, `add_at_end`, `reverse` or `comb`.

`tinyML_ulex` is an example of how to use `ulex` as a lexer before parsing with `dypgen`. The makefile requires `findlib`. Note that this example does not use the possibilities of UTF, it is just an example of how to use `ulex` with `dypgen`. The language of `tinyML_ulex` is smaller than that of `tinyML` you can only use the following characters as tokens `[`, `]`, `|`, `::`, `;`, `<`, `>`, `@`.

A Acknowledgements

Christophe Raffalli: ideas for self-extensibility and disambiguation with the system of priorities and the relation between priorities. Dypgen began as a school work supervised by Christophe Raffalli in 2005. Pierre Hyvernat made the documentation generation script.

The primary goal of dypgen is to be a convenient and useful tool for OCaml developers. To achieve this, dypgen introduces new ideas as well as gathers and implements other ideas found in other projects.

Previous versions of dypgen used Martin Bravenboer and Eelco Visser algorithm of parse table composition, adapted to handle dypgen's system of priorities and local extensibility. (it is not used anymore in this version of dypgen instead an algorithm that computes a new LR(0) table from scratch is used, which is easier for me to maintain)

[1] Martin Bravenboer and Eelco Visser, Parse Table Composition: Separate Compilation and Binary Extensibility of Grammars, 2007.

http://www.stratego-language.org/Stratego/ParseTableComposition

The idea of merge functions as well as some ideas for the GLR algorithm are borrowed from the following technical report about the parser generator Elkhound from Scott McPeak:

[2] Scott McPeak. Elkhound: A fast, efficient GLR parser generator. Technical Report CSD-02-1214, University of California, Berkeley, December 2002.

http://www.cs.berkeley.edu/s̃mcpeak/elkhound

B Migration from `ocamlyacc`

`dypgen` takes a file ending with `.dyp` as input and generates a `.ml` file and a `.mli` file. The frame of an input file for `dypgen` is somewhat similar to an input file for `ocamlyacc`. The syntax differs in the following points:

- The header and trailer codes are placed between braces {} (instead of `%{%}` for the header in `ocamlyacc`).
- The keywords `%right`, `%left` and `%nonassoc` do not exist, precedence and associativity assigned to symbol is not implemented yet. Ambiguities are managed by other means.
- The entry points are declared with their type like: `%start <int> main`, with one keyword `%start` for each entry point. The type is not mandatory.
- When tokens are declared the type statement only applies to the following token and a type statement can be stated anywhere on a line beginning with `%token`, provided it is followed by a token name. For instance:
  `%token BAR <string> UIDENT COMMA <string> LIDENT COLON`
  is the declaration of `BAR`, `COMMA` and `COLON` as tokens with 0 argument and `UIDENT` and `LIDENT` as tokens with a `string` as argument. Tokens do not need to be declared when using dypgen's lexer generator.
- There is no special symbol `error` for rules.
- There is no '`;`' between rules.
- To avoid identifier collision identifiers beginning with `__dypgen` should not be used and the name of your non terminals should not begin with `dypgen__`.
- The parser must be linked against the library `dyp.cma` (or `dyp.cmxa`) which is found in the directory `dyplib`.

C A complete example of grammar extension

The following example is implemented in the directory `demos/demo`. It is a small language with integers, pairs, constructors and variable names. The program parses the input and then prints it. If one enters the input `List(1,List(2,Nil))` the program outputs `= List(1,List(2,Nil))`. The language is somewhat extensible, with the following construction: `define lhs:=` ***`rhs`*** `=` ***`expression`*** `in` where `lhs` is the left-hand side non terminal of the rule the user wants to add, ***`rhs`*** is the right hand side of this new rule, ***`expression`*** is the expression which will be yielded when a reduction by this new rule occurs. Here is an example of introduction of a specific syntax for lists:

```
define list_contents := expr(x) = List(x,Nil) in
define list_contents := expr(x) ";" list_contents(y) = List(x,y) in
define expr := "[" "]" = Nil in
define expr := "[" list_contents(x) "]" = x in
define expr := expr(x) "::" expr(y) = List(x,y) in
[1;2;3]
```

The output is

```
= List(1,List(2,List(3,Nil)))
```

The example is made of 3 files: `parse_tree.ml`, `parser.dyp` and `demo.ml`.

```
(* parse_tree.ml *)

type expr =
  | Lident of string
  | Int of int
  | Pair of (expr * expr)
  | Cons of string * (int * (expr list))

type rhs = Token of string | Nt of (string * string)

let rec str_expr exp = match exp with
  | Int i -> string_of_int i
  | Pair (a,b) -> "("^(str_expr a)^","^(str_expr b)^")"
  | Cons (cons,(0,_)) -> cons
  | Cons (cons,(1,[o])) ->
      cons^"("^(str_expr o)^")"
  | Cons (cons,(2,[o1;o2])) ->
      cons^"("^(str_expr o1)^","^(str_expr o2)^")"
  | Lident x -> x
  | _ -> failwith "str_expr"

module String_map = Map.Make(String)

let rec substitute env expr = match expr with
  | Int i -> Int i
```

```
  | Lident s ->
      begin try String_map.find s env
      with Not_found -> Lident s end
  | Pair (a,b) -> Pair (substitute env a,substitute env b)
  | Cons (c,(n,l)) ->
      Cons (c,(n,(List.map (substitute env) l)))
```

This file declares the two types associated with the two non terminals `expr` and `rhs`. `str_expr` prints expressions and `substitute env expr` substitutes in `expr` the variables names by the expressions which they are bound to in `env` if they are present in `env`. This is used in the parser to define the action associated with a new rule.

```
/* parser.dyp */

{ open Parse_tree
open Dyp

let string_buf = Buffer.create 10

let a_define_in dyp (s,ol,e) =
  let f o =
    match o with
      | Nt (s,_) -> Non_ter (s,No_priority)
      | Token s -> Regexp (RE_String s)
  in
  let rule  = s,(List.map f ol),"default_priority",[] in
  let action = (fun _ avl ->
    let f2 env o av = match o with
      | Nt (_,var_name) -> String_map.add var_name av env
      | _ -> env
    in
    let f3 av = match av with
      | Obj_expr exp -> exp
      | _ -> Int 0
    in
    let avl = List.map f3 avl in
    let env = List.fold_left2 f2 String_map.empty ol avl in
    Obj_expr (substitute env e), [])
  in rule, action
}

%start main

%lexer

let newline = ('\010' | '\013' | "\013\010")
let blank = [' ' '\009' '\012']
```

```
let lowercase = ['a'-'z' '\223'-'\246' '\248'-'\255' '_']
let uppercase = ['A'-'Z' '\192'-'\214' '\216'-'\222']
let identchar =
  ['A'-'Z' 'a'-'z' '_' '\192'-'\214' '\216'-'\246' '\248'-'\255' '\'' '0'-'9']

rule string = parse
  | '"' { () }
  | _ { Buffer.add_string string_buf (Dyp.lexeme lexbuf);
      string lexbuf }

main lexer =
newline | blank + ->
lowercase identchar * -> LIDENT { Dyp.lexeme lexbuf }
uppercase identchar * -> UIDENT { Dyp.lexeme lexbuf }
'"' -> STRING { Buffer.clear string_buf;
      string lexbuf;
      Buffer.contents string_buf }

%parser

main : expr eof { $1 }

expr :
  | ['0'-'9']+ { Int (int_of_string $1) }
  | "(" expr "," expr ")" { Pair ($2,$4) }
  | UIDENT expr
    { match $2 with
        | Pair (a,b) -> Cons ($1,(2,[a;b]))
        | exp -> Cons ($1,(1,[exp])) }
  | UIDENT { Cons ($1,(0,[])) }
  | LIDENT { Lident $1 }
  | define_in expr { $2 }

define_in :
  | "define" LIDENT ":=" rhs "=" expr "in"
    @{ (), [ Add_rules [a_define_in dyp ($2,$4,$6)];
      Bind_to_cons [$2,"Obj_expr"]] }

rhs :
  | LIDENT "(" LIDENT ")" { [Nt ($1,$3)] }
  | STRING { [Token $1] }
  | LIDENT "(" LIDENT ")" rhs { (Nt ($1,$3))::$5 }
  | STRING rhs { (Token $1)::$2 }
```

This file contains the definitions of the parser and the lexer.
The reduction by the rule:

```
define_in: "define" LIDENT ":=" rhs "=" expr "in"
```

introduces a new rule. The string returned by `LIDENT` (*i.e.* `$2`) is the name of the non terminal of the left-hand side. The function `a_define_in` is called. It returns a rule and an action. The new rule is made of four values:

- The non terminal of the left-hand side, that is the string `s`.
- The list of symbols of the right-hand side of the rule, that is the result of `List.map f ol`.
- The priority returned by the rule, which is denoted by the string of its name, here it is `"default_priority"`.
- A boolean that tells whether layout characters are allowed to be read in the part of the input that is reduced when applying the rule. This is only relevant when using dypgen own lexer (which is the case here).

For each non terminal in the right-hand side `rhs`, a variable name follows in parentheses. The action code of the new rule is defined as returning the expression which follows the second `=` in which some variable names are substituted by some expressions. The variable names which appear in the right-hand side of the rule are substituted by the results yielded by the corresponding non terminals.

```
(* demo.ml *)

open Parse_tree

let string_ref = ref ""
let process_argument s =
  if s = "" then raise (Arg.Bad "missing input file name")
  else string_ref := s
let _ = Arg.parse [] process_argument "usage: demo file_name"
let _ = if !string_ref = "" then
  (print_string "usage: demo file_name\n";
  exit 0)

let input_file = !string_ref
let lexbuf = Dyp.from_channel (Parser.pp ()) (Pervasives.open_in input_file)
let prog = fst (List.hd (Parser.main lexbuf))

let s = str_expr prog
let () = Printf.printf "= %s\n" s
```

This is the main implementation file, it opens the file given as argument, parses it and prints the corresponding expression.

All the files of this example are available in the directory `demos/demo`. To test it with a test input do:

```
./demo test1.tiny
```