`dypgen` User's Manual

Emmanuel Onzon

June 23, 2007

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 2. Its main differences with other solutions are:

- This is a GLR parser. This means it can handle ambiguous grammars and output 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 `dypgen` source file.
- Ambiguities can be removed by introducing *priorities* and relations between them. This gives a very natural way to express a grammar (the example in this documentation illustrates this).
- Grammars are self-extensible, i.e. an action can modify 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 data that the user can access and modify. These mechanisms adress the problem posed by side effects with GLR parsing (see section 6).

  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.

  This 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 *partial actions*, that are semantic actions performed before the end of a rule.

Contents

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`. The complete source for this example lies in the directory `demos/calc` 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 <int> 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.

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

2 BNF grammars

A BNF grammar is a concise way of defining a language, that is set of sequence of characters. However, it is traditional and may be more efficient to define 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 set of sequence of terminal symbols (they have sometime 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).

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 to 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 to 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 parenthesis.

More details about priorities will be given in the section 5.1.

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 to 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 to) each non-terminal. Terminal symbols also can have semantics associated to 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.

5 Managing 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).

5.1 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 to 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 to 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.

When a priority is declared a corresponding Caml value of type `priority` is introduced with the same name. You should beware of identifier collision and not name other variables with the names of your priorities or `default_priority`. This value can be used in action codes. All the information about relations are stored in a structure of type `priority_data`. This structure is accessible from the action code with the mutable record field `dyp.priority_data`. The following function allows to know whether the relation holds between two priorities:

```
val is_relation : priority_data -> priority -> priority -> bool
```

`is_relation dyp.priority_data p q` returns true if `p<q` holds and false otherwise. The type of the record `dyp` is defined in the module `Dyp` of the library `dyp.cm[x]a`, see the section 10 for more information about it.
Other functions pertaining to priorities are available, see section 9 about changing the priorities at runtime.

5.2 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 parse trees yielded or to choose just one or a few of them, or to make a new tree from them. By default only one parse tree is kept: the oldest, and the other ones are discarded but the user can make it otherwise. It is possible to define a function `dyp_merge_nt` of type :

```
val dyp_merge_nt : 'a list -> 'a -> 'a list
```

Where `'a` is the type of the value returned when one reduces to the non terminal `nt`. The first argument is the list of parse trees which are the different interpretations which were yielded and kept for the non terminal `nt` for a given part of the input. The second argument is the parse tree which has been just yielded. The result is a list of parse trees which are kept as the different interpretations for the non terminal `nt` for the considered part of the input. The user can define such a function for each non terminal in the header of the parser.

For instance if one wants to keep all the parse trees for the non terminal `nt` then we can use:

```
let dyp_merge_nt ol o = o::ol
```

If one wants to keep the newest parse tree:

```
let dyp_merge_nt _ o = [o]
```

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

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 dyp_merge_expr l o2 = match l with [o1] ->
  begin match o1 with
    | Times (Plus(_,_),_) -> [o2]
    | Times (_,Plus(_,_)) -> [o2]
    | _ -> [o1]
  end | _ -> assert false
```

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

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 a generic merge function as well as the type of the global merge function is the following:

```
type merge_function : priority_data -> ((obj * priority) list) -> (obj * priority)
  -> ((obj * priority) list)
```

The type `obj` should be considered as an abstract type which cannot be destructured. If you want to define a merge function which can destructure a parse tree and to assign this merge function to several non terminals (which must return the same type) then you have to use the specific merge functions discussed above.

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 ol o = o::ol
```

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 nt1 nt2 nt3 nt4 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.

There are three predefined generic merge functions available to the user:

```
val keep_all : merge_function
val keep_oldest : merge_function
val keep_newest : merge_function
```

They keep respectively all the parse trees, the oldest and the newest. If no global merge function is defined then by default it is `keep_oldest`.

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 parse trees 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 merge in the directory `demos/forest` where a parse forest is yielded.

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 happens.

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.

5.3 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 try 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 further. Then the parser reduces `1::[2]` to an expression and thereafter `head 1::[2]` is reduced to an expression without type error.

5.4 Preventing a shift

When a reduction occurs it is possible to prevent the shift with the action code. You just have to use the following line in your action code:

```
dyp.will_shift <- false;
```

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 { dyp.will_shift <- false; action_comma $1 $3 }
```

Then there is only one way to reduce `1,2,3` to `expr` : the first one, because when the record field `dyp.will_shift` is `false` the parser will not shift the comma without reducing. And there is only one way to reduce `1,...,n` to `expr`.

6 Auxiliary data

With GLR, the parsing can follow different interpretations independantly 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 the mutable record fields `dyp.global_data` or `dyp.local_data`. If they are used, then the user must define their initial values in the header with the following references:

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

The record `dyp` is available in the action code and you can change the content of the fields `global_data` and `local_data` (the type of the record `dyp` is defined in the module `Dyp` see section 10 for more information). The initial values can also be accessed from outside the parser definition if you declare them with their type in the `.mli` file (by using `%mli`, see section 13.2). If you change them before calling the parser, then these changes will be taken into account as new initial values for `global_data` and `local_data`.

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 independantly for each alternative.

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 update `dyp.local_data` in an action which yields a non terminal `nt`, then this `dyp.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 }
b: e f   { action_3 }
e: A1    { action_1 }
f: A2    { action_2 }
c: g h   { action_6 }
g: B1    { action_4 }
h: B2    { action_5 }
d: C     { action_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` which it modifies to `local_data_1`,
- then the next action is `action_2`, it has access to `local_data_1` and can modify it to `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 changes `dyp.local_data` to the value `local_data_3`.

- The next action to be performed is `action_4`, it changes `local_data_3` to `local_data_4`,
- then `action_5` has access to this new value and can change it 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. It is changed to `local_data_6` by `action_6`.
- The next action is `action_7` which has access to `local_data_6`. It is useless to change it 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.

`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 EOL { $1 }
expr: INT                  { Int $1 }
  | IDENT                  { if is_bound dyp.Dyp.local_data $1 then Ident $1
                             else raise Dyp.Giveup }
  | LPAREN expr RPAREN     { $2 }
  | expr PLUS expr         { Plus ($1,$3) }
  | LET binding IN expr    { Let ($2,$4) }
binding: IDENT EQUAL expr
    { dyp.Dyp.local_data <- insert_binding dyp.Dyp.local_data $1 $3;
      Binding ($1,$3) }
```

If we keep all the parse trees (see merge functions section 5.2), 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
```

yields only 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`.

A function of equality between two global data `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 in order to not merge two interpretations of a given part of the input if they have different `global_data` or different `local_data`

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

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

%token <int> INT PLUS TIMES EOL
%relation pi<pt<pp
%start <int> main
%%
main: expr EOL { Printf.printf "The parser made %d reductions for this
                 interpretation.\n" dyp.global_data; $1 }
expr:
  | INT                         { dyp.global_data <- dyp.global_data+1; $1 }      pi
  | expr(<=pp) PLUS expr(<pp)  { dyp.global_data <- dyp.global_data+1; $1 + $3 } pp
  | expr(<=pt) TIMES expr(<pt) { dyp.global_data <- dyp.global_data+1; $1 * $3 } pt
```

For instance we parse `5*7+3*4`, when `4` is reduced, `dyp.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 `dyp.global_data` that `x` is of type float and then when we reduce `x+1` we have this information still stored in `dyp.global_data` which is accessible from the action code. And we can detect the type incompatibility whithout having to parse more input.

7 More about actions and rules

7.1 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`. This is useful when one wants to add a rule at runtime with a keyword which is not recognized specifically as a keyword by the lexer but as an identifier.

7.2 Partial actions

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

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

The value returned by the partial action is accessible to the final action as if the partial 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`.

For the parser the rule :

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

does not exist. `dypgen` splits this rule in the two following:

```
expr: dypgen__nt_0 IN expr
dypgen__nt_0: LET IDENT EQUAL expr
```

As a consequence if you modify `local_data` in the partial 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 =
  dyp.local_data <- insert_binding dyp.local_data name exp;
  Binding (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` assigned by `binding` called by the partial action.

It is possible to insert several partial action in a rule, but not at the same position in the right-hand side.

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

creates the following rules:

```
nt1: dypgen__nt_1 TOK3
dypgen__nt_1: dypgen__nt_2 nt4 TOK2
dypgen__nt_2: dypgen__nt_3 nt2 nt3
dypgen__nt_3: TOK1 nt2
```

As a consequence of these new rules, if `local_data` is changed by `pa_1`, these changes stay accessible until `pa_2` is performed (`pa_2` not included). Note that the names of the generated new non terminals are unique, the ones stated here may be inaccurate.

Another consequence is that if a rule exists two times in the parser definition, one time with partial actions and another time whithout partial action, or both times with partial 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 partial action.

7.3 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 brackets after the symbol name and 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  { ... }
```

The value returned by a partial action can also be matched. You can write:

```
nt0: TOK1 nt1 { partial_action }[pattern] nt2 TOK2 nt3 { action }
```

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

7.4 Nested rules

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

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

The pattern in brackets after the list of nested rules is optionnal.

8 Dynamic changes to the grammar

8.1 Adding and removing rules

Dynamic changes to the grammar are performed by action code. To add rules one uses the mutable record field `dyp.add_rules` and to remove rules the mutable record field `dyp.remove_rules`.

```
mutable add_rules : (rule * (dypgen_toolbox -> obj list -> obj)) list;
mutable remove_rules : rule list
```

Where `dypgen_toolbox` is the type of the record `dyp` (see section 10), and the type `obj` is explained a bit further. To add several rules and their respective actions to the grammar, the user assigns the list of the corresponding couples `(rule, action)` to `dyp.add_rules`.

The type `rule` is defined as follows :

```
type token_name
type non_ter
type 'a pliteral =
  | Ter of token_name
  | Non_ter of 'a * non_terminal_priority
type lit = (non_ter * non_terminal_priority) pliteral
type rule = non_ter * (lit list) * priority
```

In the type `rule`, `non_ter` is the non terminal of the left-hand side of the rule, `lit list` is the list of symbols in the right-hand side of the rule. These types are part of the module `Dyp` which is not open by default.

For each token `Token` there is one `token_name` value which is bound to the variable `t_Token`. Non terminals are accessible with their names which are variable names. All these values (tokens names and non terminals names) are encapsulated in a module named `Dyp_symbols`. For instance if you have a non terminal `expr` and you want to use it to build a new rule, then you would use `Dyp_symbols.expr`. It is a value of type `non_ter`.

The type `non_terminal_priority` is :

```
type non_terminal_priority =
  | No_priority
  | Eq_priority of priority
  | Less_priority of priority
  | Lesseq_priority of priority
  | Greater_priority of priority
  | Greatereq_priority of priority
```

Which is also part of the module `Dyp`.

The type `obj` is a sum of constructors with each constructor corresponding to a terminal or 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 to 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 to 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 waiting 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 non terminals to have the same constructor then you can use the directive:

```
%constructor Cons %for nt1 nt2 ...
```

where `Cons` is the name of the common constructor you want to assign for all these non terminals. Of course all these non terminals need to return values of the same type. And even if you decide to use polymorphic variants (see below) 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 non terminal in the initial grammar but that may be used for new non terminals when new rules are added dynamically (see section 8.4). For instance:

```
%constructor Cons1 Cons2 Cons3
```

Note that `obj` is actually a type constructor rather than a type. The types of values returned by the tokens and the non terminals 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.

8.2 Example

```
{ open Dyp
open Dyp_priority_data

let rule_plus = (Dyp_symbols.expr, [
  Non_ter (Dyp_symbols.expr,No_priority);
  Ter Dyp_symbols.t_PLUS;
  Non_ter (Dyp_symbols.expr,No_priority)
  ], default_priority)
(* define the rule : E -> E + E *)

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

%token <int> INT PLUS AMPERSAND EOF
%start <int> main
%%
main : expr EOF    { $1 }

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

a : AMPERSAND { dyp.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 PLUS expr` to the initial grammar which does not contain it. And the action code associated to 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, 'expr) obj =
  | Obj_AMPERSAND
  | Obj_EOF
  | Obj_INT of (int)
  | Obj_PLUS
  | Obj_a of 'a
  | Obj_expr of 'expr
  | Obj_main of int
```

Note that with a partial action we can use alternatively the following grammar:

```
main : expr EOF    { $1 }
expr :
  | INT            { $1 }
  | AMPERSAND { dyp.add_rules <- [(rule_plus, action_plus)] } expr { $3 }
```

Now the type constructor `obj` becomes:

```
type ( 'dypgen__nt_0, 'expr) obj =
  | Obj_AMPERSAND
  | Obj_EOF
  | Obj_INT of (int)
  | Obj_PLUS
  | Obj_dypgen__nt_0 of 'dypgen__nt_0
  | Obj_expr of 'expr
  | Obj_main of int
```

8.3 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 *)
}

%token <int> INT PLUS AMPERSAND LPAREN RPAREN EOF
%start <int> main
%%
```

```
main : expr EOF    { $1 }

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

a : AMPERSAND { 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 PLUS 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 close to this one.

8.4 Adding new non terminals

To add a new non terminal to the grammar from the action code, one uses the following functions:

```
dyp.add_nt : string -> string -> Dyp.non_ter
dyp.find_nt : string -> Dyp.non_ter * string
```

When new rules with new non terminals are added, each new non terminal must be added to the grammar as a string, paired with another string representing the constructor of the values returned by this non terminal.

```
dyp.add_nt nt cons
```

is used to insert a new non terminal which key string is `nt` and which associated constructor has the key string `cons`. The function returns a fresh value of type `non_ter` that can be used to build rules. 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.

Any non terminal of the initial grammar is bound with its corresponding string. If the string `nt` is already bound to a non terminal (from the initial grammar or that has been added subsequently) then there are two possible outcomes:

- The string `cons` is the same as the one that is already bound to this non terminal, then `dyp.add_nt` returns the value of type `non_ter` that was already bound `nt`.

- The string `cons` is different from the one that is already bound to this non terminal, then `dyp.add_nt` raises:

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

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

`dyp.find_nt nt_key` returns a couple `(nt,cons)` where `nt` is the value of type `non_ter` associated to the string `nt_key` and `cons` is the string of the associated constructor, if it is bound and it raises `Not_found` otherwise.

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 B. It describes a small language that is somewhat extensible.

9 Changing dynamically priority data

The following functions are part of the module `Dyp` and makes possible to change the priorities and their relations.

```
val insert_priority : priority_data -> string -> (priority_data * priority)
val find_priority : priority_data -> string -> priority
```

`insert_priority` creates a new priority and inserts it in the priority data structure. The user gives a key which is a `string` and the original priority data structure. The function returns the priority data structure with the new priority added and the value of type `priority` corresponding to the new priority. For instance, to insert a new priority with key `"new_prio"`, write the following inside your action code:

```
let newprio_data, new_prio = insert_priority dyp.priority_data "new_prio" in
dyp.priority_data <- newprio_data;
```

```
val find_priority : priority_data -> string -> priority
```

`find_priority priodata key` returns the priority associated to the string `key` in the priority data structure `priodata`.

```
val set_relation : priority_data -> bool -> priority -> priority ->
  priority_data
```

`set_relation priodata b p q` returns a priority data structure which is `priodata` with the following change: if `b` is `true` then `p<q` holds, if it is false `p<q` does not hold.

```
val update_priority : priority_data -> (priority * priority * bool) list -> priority_data
```

`update_priority priodata ppl` returns the priority data structure `priodata` with the following changes: for each triple `(p,q,true)` in `ppl`, the relation `p<q` holds and for each triple `(p,q,false)` the relation `p<q` does not hold.

```
val add_list_relations : priority_data -> (priority list) -> priority_data
```

`add_list_relations priodata pl` returns the priority data structure `priodata` with the following change: if `pl` is `[p1;...;pn]` then `p1<...<pn` holds in the sense defined in section 5.1.

10 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 ('obj,'data,'local_data) dypgen_toolbox = {
  mutable global_data : 'data;
  mutable local_data : 'local_data;
  mutable priority_data : Dyp.priority_data;
  mutable add_rules : (Dyp.rule * (
    ('obj,'data,'local_data) dypgen_toolbox -> 'obj list -> 'obj)) list;
  mutable remove_rules : Dyp.rule list;
  mutable will_shift : bool;
  mutable next_state : out_channel option;
  mutable next_grammar : out_channel option;
  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;
  add_nt : string -> string -> Dyp.non_ter;
  find_nt : string -> (Dyp.non_ter * string);
  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 8.1.

11 Names conflicts

To avoid names conflicts you should not use identifier beginning with `__dypgen_` and take into account the names that are listed in this section.

This is the list of the names available in the code of the parser:

```
type token

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
...

type ('obj,'gd,'ld) dypgen_toolbox

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
...

module Dyp_symbols :
sig
  val nt1 : int
  val nt2 : int
  ...
  val t_TOKEN_1 : int
  val t_TOKEN_2 : int
  ...
  val get_token_name : token -> int
  val str_token : token -> string
end

module Dyp_priority_data :
stig
  val priority_data : Dyp.priority_data
  val default_priority : Dyp.priority
  val prio_1 : Dyp.priority
  val prio_1 : Dyp.priority
  ...
```

```
end
```

In addition the following module names are used:

```
module Dyp_symbols_array
module Dyp_parameters
module Dyp_runtime
module Dyp_engine
module Dyp_aux_functions
```

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

```
val dypgen_verbose : int ref
type token_name = int
type non_ter = int
type 'a pliteral =
  | Ter of token_name
  | Non_ter of 'a
type priority
type non_terminal_priority =
  | No_priority
  | Eq_priority of priority
  | Less_priority of priority
  | Lesseq_priority of priority
  | Greater_priority of priority
  | Greatereq_priority of priority

type priority_data
val empty_priority_data : priority_data
val is_relation : priority_data -> priority -> priority -> bool
val insert_priority : priority_data -> string -> (priority_data * priority)
val find_priority : priority_data -> string -> priority
val set_relation : priority_data -> bool -> priority -> priority ->
  priority_data
val update_priority : priority_data -> (priority * priority * bool) list ->
  priority_data
val add_list_relations : priority_data -> (priority list) -> priority_data

type lit = (int * non_terminal_priority) pliteral
type rule = non_ter * (lit list) * priority

type ('obj,'data,'local_data) dypgen_toolbox = {
  mutable global_data : 'data;
  mutable local_data : 'local_data;
  mutable priority_data : priority_data;
  mutable add_rules : (rule * (
    ('obj,'data,'local_data) dypgen_toolbox -> 'obj list -> 'obj)) list;
  mutable remove_rules : rule list;
  mutable will_shift : bool;
  mutable next_state : out_channel option;
  mutable next_grammar : out_channel option;
  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;
  add_nt : string -> string -> non_ter;
  find_nt : string -> non_ter * string;
  print_state : out_channel -> unit;
  print_grammar : out_channel -> unit;
}


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

type 'obj merge_function = 'obj list -> 'obj -> ('obj list)
type 'obj merge_map

val keep_all : 'a list -> 'a -> 'a list
val keep_oldest : 'a list -> 'a -> 'a list
val keep_newest : 'a list -> 'a -> 'a list

module type Dyp_parameters_type
module Dyp_special_types
module Make_dyp
```

12 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` (in the `demo` directory)...

13 Other features

13.1 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`.

```
  dyp.next_state <- Some out_channel;
```

will print the next state of the automaton (the one where the parser goes after the current reduction)

```
  dyp.next_grammar <- Some out_channel;
```

will print the grammar after the current reduction (and possible changes).

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 then the beginning and the end of the part of the input is given.

13.2 Adding code to the interface of the parser

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

13.3 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.

13.4 Using another lexer than `ocamllex`

To use another lexer than `ocamllex` use the option `--lexer other` with `dypgen`. The interface of the entry functions are then changed to:

```
val entry : ('a -> token) -> 'a -> (int * Dyp.priority) list
```

Whereas by default it is:

```
val entry : (Lexing.lexbuf -> token) -> Lexing.lexbuf -> (int * Dyp.priority) list
```

Note that the functions returning the positions of the lexer in the action code do not return relevant information with lexers other than `ocamllex`.

13.5 Position of the lexer

When `ocamllex` is used the following functions are available:

```
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.

13.6 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. Then you use the options `--pv-token` and `--pv-obj`. With `--pv-token`, the `token` type is a sum of polymorphic variants, and with `--pv-obj` the type constructor `obj` uses polymorphic variants.

13.7 Automaton options

By default the enforcement of the priorities is embedded into the automaton. This makes the parsing faster than if they were enforced at parsing time but the automaton is bigger and longer to generate. You can not embed the priorities in the automaton and make them enforced at parsing time with the option `--prio-pt`.

14 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 7.3), and `calc_nested` uses nested rules.

`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 *.

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

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

global_data and local_data are example of using global_data and local_data. local_data_partial_ac is the same as local_data except that it uses a partial action.

position is a small example using the functions dyp.symbol_start, . . . which returns the position of a part of the input which is reduced to a given non terminal.

demo is the example of appendix B.

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 using the following characters as tokens [,], |, ::, ;, <, >, @ and the non terminal corresponding to expressions. 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.

A Comparison with 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.
- 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.

- 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.
- `dypgen` does not handle cyclic grammars (i.e. when a non terminal can derive itself). When the parser is generated, there is no warning if the grammar is cyclic. But the parser fails with an error when used.
- The parser must be linked against the library `dyp.cma` (or `dyp.cmxa`) which is found in the directory `dyplib`.

B A complete example of grammar extension

This example 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)))
```

A distinction is made between a token inside the right-hand side of the construct `define ... in` and the same token outside of this right-hand. This distinction is made by the lexer. For instance, for the character '`[`' it returns `Token "["` if it is between `:=` and `=`, and it returns `LBRACK` otherwise.

The example is made of 4 files : `parse_tree.ml`, `parser.dyp`, `lexer.mll` 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 Ordered_string =
struct
  type t = string
  let compare = Pervasives.compare
end

module String_map = Map.Make(Ordered_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 to 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 to new a rule.

```
/* parser.dyp */

{ open Parse_tree
open Dyp_symbols
open Dyp_priority_data
open Dyp

let () = dypgen_verbose := 1

let get_token_name s = match s with
```

```
  | "[" -> t_LBRACK
  | "]" -> t_RBRACK
  | "::" -> t_COLONCOLON
  | ";" -> t_SEMICOLON
  | _ -> failwith "get_token_name"

let a_define_in dyp (s,ol,e) =
  let lhs,_ = dyp.find_nt s in
  let f o =
    match o with
      | Nt (s,_) -> Non_ter (fst (dyp.find_nt s),No_priority)
      | Token s -> Ter (get_token_name s)
  in
  let rule  = lhs,(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
}

%token LPAREN RPAREN COMMA <string> UIDENT <string> LIDENT <int> INT DEFINE IN EQUAL COLONI

%start <Parse_tree.expr> main

%%

main : expr EOF { $1 }

expr :
  | INT { Int $1 }
  | LPAREN expr COMMA expr RPAREN { 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 COLONEQUAL rhs EQUAL expr IN
    { let _ = dyp.add_nt $2 "Obj_expr" in
      dyp.add_rules <- [a_define_in dyp ($2,$4,$6)] }

rhs :
  | LIDENT LPAREN LIDENT RPAREN { [Nt ($1,$3)] }
  | TOKEN { [Token $1] }
  | LIDENT LPAREN LIDENT RPAREN rhs { (Nt ($1,$3))::$5 }
  | TOKEN rhs { (Token $1)::$2 }
```

The reduction by the rule :

```
define_in : DEFINE LIDENT COLONEQUAL rhs EQUAL 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. It is added as a new non terminal with the line :

```
let _ = dyp.add_nt $2 in
```

Then the function `a_define_in` is called. It returns a rule, an action and a priority. To construct the new rule, `dyp.find_nt` and `get_token_name` are used. Non terminals are refered to by a string `s`, `dyp.find_nt s` returns the corresponding value of type `non_ter`. `get_token_name s` returns the value of type `token_name` corresponding to the string `s`.

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.

For information about `dypgen_verbose`, which is stated at the beginning of the parser definition, see section 13.1.

```
{ (* lexer.mll *)

open Parser
let lex_define = ref false }

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 token = parse
```

```
  | newline
      { token lexbuf }
  | blank +
      { token lexbuf }
  | "define" { DEFINE }
  | "in" { IN }
  | lowercase identchar *
      { LIDENT(Lexing.lexeme lexbuf) }
  | uppercase identchar *
      { UIDENT(Lexing.lexeme lexbuf) }
  | "("  { LPAREN }
  | ")"  { RPAREN }
  | "=" { lex_define := false; EQUAL }
  | ":=" { lex_define := true; COLONEQUAL }
  | "[" { if !lex_define then TOKEN "[" else LBRACK }
  | "]" { if !lex_define then TOKEN "]" else RBRACK }
  | "::" { if !lex_define then TOKEN "::" else COLONCOLON }
  | ";" { if !lex_define then TOKEN ";" else SEMICOLON }
  | ['0'-'9']+ as lxm { INT(int_of_string lxm) }
  | "," { COMMA }
  | eof { EOF }
```

This is the lexer. Some strings are special, they do not always yield the same token, they are the ones the user can use in the `define ... in` construct. The ref `lex_define` is a flag which tells the lexer whether it is between `:=` and `=` or not. In the first case, any special string `s` would yield `TOKEN s`, in the second case it would yield its corresponding regular token (like `SEMICOLON` for '`;`').

```
(* 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 = Lexing.from_channel (Pervasives.open_in input_file)
let prog = fst (List.hd (Parser.main Lexer.token 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
```