parser/tdop.mli

Interface for module Tdop

This module helps implementing TDOP (top-down operator precedence) parsers. TDOP parsers (also called Pratt parsers) were invented by Vaughan Pratt in 1973. They are practical, efficient, simple to understand and simple to implement, and can be dynamically extended at runtime; their only drawback is that they are not very well-known.

The core idea of TDOP is to combine "recursive descent" parser with "operator precedence" in a single framework; and to work by associating parsing functions to tokens (and not to rules as done in BNF).

Main parsing function and token position

The module works by defining a "main" parsing function `parse`, that will call auxiliary parsing functions associated to the tokens, depending on their position:

- The first token encountered when `parse` is called is in *prefix* position.
- If at a position `parse` could have returned something, but continues, then the first token encountered after `parse` could have returned is in *infix* position.

For instance, suppose we are parsing an expression (i.e. `parse` parses an "expression"). In the string `-3 + 4 × -5 - 22`, the left-most `-` appears in prefix position. It is associated to a parsing function requiring another expression, so `3` is also in prefix position. After `3` `parse` could return, but continues with `+`, so `+` is in infix position. `+` is associated to a parsing function requiring another expression, so `4` appears in prefix position, etc. At the end in this expression, all the numbers and the first two `-` are in prefix position, while `+`, `×`, and the last `-` are in infix position.

Note that the name "infix" does not imply a symmetry: in the regexp `[a-z]+`, `+` is also in infix position; in the statement `(3+4):Int`, `:` is in infix position, even if its left side is an expression and its right side a type. So "infix" also encompass "postfix" and other situations.

Interface

The main parsing function is called `parse`; it takes a stream, and a right binding power (defined below), as arguments.

The `define_prefix` and `define_infix` functions allow to associate to each keyword a parsing function. Parsing functions are given a stream; the prefix or infix token is left in the stream when the custom parsing function is called. This allows, for instance, the same parsing function to be used for different tokens. The `define_infix_{left|right}_associative` functions are special cases of `define_infix`, explained below. This dynamic interface allows to dynamically extend the parser by registering new token/parsing function association; an application is to allow a source file to extend its syntax. It also breaks recursivity, which allow the parsing functions to call `parse`.

The dynamic interface cannot work for tokens that carry an information, such as ints and string: one cannot associate a parsing function to "all ints" using a hash table. Thus for non-keyword arguments, the parsing functions are given statically as arguments to a functor. The interface assumes that only keyword tokens can be used as infix.

The argument to the functor also define the type of the "semantic values", i.e. the objects returned by the parsing functions.

Binding power, operator precedence, and associativity

In TDOP, tokens are associated with two binding powers: the left binding power, and the right binding power. If a substring has the form `AEB`, where `A` and `B` are tokens and `E` an expression that can be parsed by `parse`, then:

- If the left binding power of `B` is strictly higher than the right binding power of `A`, then `E` is associated to `A`;
- Else, `E` is associated to `A`.

As a convention, we chose that in case of equality, `E` is associated to `A`.

Thus, "left binding power" of `B` really means "the binding power of `B` that applies to the expression `E` which is to the left of `B`" (and respectively for the "right binding power").

Here are some example uses of these binding powers.

Classical operator precedence

In mathematics, × takes precedence over +: `2+3×4` is parsed as `2+(3×4)`; while `2×3+4` is parsed as `(2×3)+4`. If we give to × left and right binding powers of 100, and to + left and right binding powers of 90, then in both cases when parsing 3, the token × will take precedence over +. Unary prefix operators like - should be given a big right binding power, to take precedence over + and *.

Left associativity

The `-` operator is left-associative: `1-2-3` must be read as `1-(2-3)`, not as `(1-2)-3`. If we give to `-` equal binding powers of 90, then when parsing 2 the leftmost `-` will take precedence over the rightmost one; this is because of our convention that when binding powers are equal, the expression is associated to the leftmost token.

Right associativity

The → operator for types is right-associative: `Int → Float → Bool` is actually `Int → (Float → Bool)`. TDOP handles right-associative operators with equal ease: we associate to → a left binding power of 80, and a right binding power of 79. Now when parsing `Float` the rightmost → token takes precedence, which makes →

right-associative.

Function calls

In classical mathematics, function call is denoted with `(`, i.e. the use of the `(` operator that follows a function name, as in `2×f(3+4)`. Here, `(` must be given a high left-binding power (to bind more tightly than `×`), and a low right-binding power (to bind less tightly than `+`); for instance `120` and `0`.

Interface: affecting a left and right binding power

The left binding power of a token is provided together with the custom parsing function, using one of the `define_infix_*` functions. The right binding power is passed as an argument to the `parse` function: the idea is to associate to each token an infix and/or prefix *handler*, which is a custom parsing function; this parsing function will call `parse` with the correct right binding power.

Note that there are predefined functions to help defining left-associative and right-associative operators, so there is no need to write custom parsing functions for these common cases.

In L, left-associative and right-associative operators are defined using `define_infix_{left|right}_associative`, and we defined 0 to be the initial right binding power for the first call of the `parse` function; when calling `parse` from a custom parsing functions, the right binding power should generally be 0.

Using separation levels

In addition to "normal" TDOP parsing, this parser also takes advantage of the separation levels of token provided by the lexer: the idea is that the same token used with different separations can be considered as if there were two different tokens, and can have different left and right binding powers, and parsing functions.

So one can view the use of separations as replacing the `Token.token → handler` and `Token.token → left_binding_power` maps of the normal TDOP interface, by `Token.token × separation →  handler` and `Token.token × separation → left_binding_power` maps. However, most of the time tokens with different separations will still be bound to the same binding powers and parsing function. Thus, to make the token/parsing function more efficient and partical, using separations is in fact done using a "secondary dispatch" – there is one dispatch for the token, and a second dispatch for the separation of this token. This amounts to currying the maps, i.e. we really have a `Token.token → separation →  left_binding_power` map. Here are some concrete recipes:

- Compared to "normal" TDOP parsing, `define_infix` functions takes a `Token.with_info → left_binding_power_provider` instead of just a `left_binding_power`. The `Token.with_info` contains the separation information. Thus, if you want to completely ignore the separation information and perform "normal" TDOP parsing, you just have to pass `(fun _ → binding_power)` as a `binding_power_provider`.
- To consider tokens with different separation information differently, you must rely on the fact that the `infix_handler` and `prefix_handler` receive streams that still contain the token used to do the dispatch. The handler can dispatch and perform different actions according to the separation information in this token: make different semantic actions, or give different right binding powers. Combined with the dispatch offered by using a custom `left_binding_power_provider`, one can for instance consider `x.y` and `x . y` entirely differently, as in Haskell.
- If you want to forbide certains spacings (for instance, presence of whitespace before a "(" token), you can

raise errors (or warnings) in the `left_binding_power_provider` or the parsing functions for some separations.
- If you want to consider newline as a statement separator, you need to prevent infix tokens appearing after a newline to be parsed as infix. For instance, in "x := 3 - 4", there are two consecutive statements. This can be achieved by giving a low left binding power to infix tokens separated by a newline.

  For instance, in `L` we call the `parse` function with a `right_binding_power` of 0. Then we just need to give a negative `binding_power` to infix tokens that follow a newline to make them appear in different statements.

Additional notes

Using TDOP as part as a larger parser

The `parse` function provided by the TDOP parser is a regular parsing function, and can be integrated as a part of any hand-written parser; including another TDOP parser. Several instances of the module can indeed be used simultaneously, for instance, one for the grammar describing the expressions, and one for the grammar describing the types.

The interface of this module, with dynamic registration of parsing functions, guarantees that there will be no problem of self- or mutual- recursion using a TDOP parser: you can always begin by defining the TDOP modules, and then use the `parse` functions.

Efficiency

Top-down parsing using recursive descent parsers are often slow because the parser recurses on the rules, and expressing operator precedence using only rules make the parser call a lot of function before finding the correct rule that allows to parse a token. Moreover, talking about precedence levels makes the grammar easier to explain to understand. Because of these reasons, recursive descent parsers often come in combination with a more efficient "operator precedence" parser.

In TDOP, the actions are associated to the tokens, not to the rule, which makes the time complexity of the parsing linear in size of the stream of tokens; more over it makes operator precedence and recursive descent parsing are nicely integrated in a single simple framework.

References

The seminal paper that invented Pratt parsers was seen in Pratt, Vaughan. "Top down operator precedence." Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (1973). The subject was studied more in depth (in particular, how a TDOP-based formal language could be used to describe a language and implement a parser, as an alternative to yacc and BNF) by Van De Vanter, Michael L. "A Formalization and Correctness Proof of the CGOL Language System." (Master's Thesis). MIT Laboratory for Computer Science Technical Report MIT-LCS-TR-147 (Cambridge, MA). 1975.

```
module type BASE_PARSER =
sig
   (* The type of the semantic values, returned by the semantic actions. *)
   type t
```

Prefix handlers for base cases.

```
  val int_handler : int → Token.Stream.t → t
  val ident_handler : Token.ident → Token.Stream.t → t
end

module Make : functor (Base_parser : BASE_PARSER) →
sig
```

A prefix handler is a custom parser that is called whenever a specific keyword is encountered in prefix position. The keyword is still in the stream, and must be consumed by the handler.

```
  type prefix_handler = Token.Stream.t → Base_parser.t
```

An infix handler is a custom parser that is called whenever a specific keyword is encountered in "infix" position. The keyword is still in the stream, and must be consumed by the handler function. The part at the left of the keyword is already parsed, and given as the second argument.

```
  type infix_handler = Token.Stream.t → Base_parser.t → Base_parser.t
```

The binding power is a priority level for a token used as an infix operator. The higher it is, the stronger the operator binds the argument. For instance, 3 + 3 * 2 + 4 is parsed as 3 + (3 * 2) + 4 because the left binding power of '*' is higher than the right binding power of '+'.

Said differently, if the `parse` function is called with a right binding power of `x`, then parsing will stop before tokens whose left binding power is ≤ `x`.

```
  type left_binding_power = int
  type right_binding_power = int
```

`left_binding_power_provider` changes the binding power according to separation level. The `left_binding_power_provider` can also return a parse error, if a token is used with incorrect separation.

```
  type left_binding_power_provider =
    Token.With_info.t → left_binding_power
```

Parses the stream with a right binding power x, e.g. stops the parse as soon as it encounters a token whose `left_binding_power` is ≤ `x`. Uses the prefix and infix handlers that have been defined.

```
  val parse : Token.Stream.t → right_binding_power → Base_parser.t
```

Associates a prefix handler to a keyword.

```
  val define_prefix : Token.token → prefix_handler → unit
```

Associates a left binding power and infix handler to a keyword.

```
val define_infix : Token.token → left_binding_power_provider → infix_handler → unit
```

Most infix operators are left-associative or right-associative operators, i.e. to parse the right part of the token one just have to call the tdop parser again. These functions handle these cases.

The first argument is the parsed token to which the function is attached (in infix or prefix position). The second is the binding power of the operator. The last is a function that, given the object resulting from parsing the left part, and the one obtained from parsing the right part, gives the object obtained by combining them.

```
  val define_infix_left_associative :
    Token.token → left_binding_power_provider →
```

```
(Token.With_info.t → left:Base_parser.t → right:Base_parser.t → Base_parser.t) →
     unit
   val define_infix_right_associative :
     Token.token → left_binding_power_provider →

(Token.With_info.t → left:Base_parser.t → right:Base_parser.t → Base_parser.t) →
     unit
```

Returns the left_binding_power assigned to an infix operator. Unregistered tokens have a binding power of zero.

```
   val left_binding_power : Token.With_info.t → left_binding_power

end
```
