Flat Tokenizer

Gaël de Chalendar

07/04/2010

1. Rationale

The original tokenizer written by Nautitia was performant concerning its tokenization speed, but the automaton on which it was based was too complex and described in a terribly too verbose XML format. The automaton used subautomatons silently executed when entering some states. These subautomatons returning a status possibly used by the transitions of the states to chose to fire or not. This was already complex. Furthermore, the various transitions were able to advance on the text or not depending on the associated actions. Finally, the XML file describing the automaton was terribly verbose: for example in Franch, more than 900 lines for the automaton itself, a single transition being described by up to 20 lines and more than 1400 lines to describe 211 characters and 39 char classes.

So, each time we had to correct a tokenization error, it took at least half a day to understand again the format, to dig into the automaton and to make the changes to work. So, I decided to partly rewrite the tokenizer to ease the maintenance of the existing automatons and make easier the creation of a new automaton for a new language.

2. Principles

Two main principles directed the rewrite:

- use a simple description format, easy to read for the human and easy to parse for the machine. The resulting French data uses 256 lines for the automaton and 251 for the chars chart;
- use an automaton simple to understand with a regular and systematic behavior: the new automaton systematicaly advance by one char (code point) on a successful transition and applies one and only one action at a standardized point.

Both file formats are parsed using the Boost.Spirit library which allows to define BNF like grammars directly in C++ and to attach code to each part of the grammar. This allows to write grammars and exploit them very efficiently in our C++ code. This allows also to write little validation tools to check the validity of the parsed files. Current tools for chars chart and automaton check just verify the basic syntax but it will be quite easy to add more semantic check, for example to ensure that a state name used is effectively used.

Both file formats described below support the use of comments defined by a '#' and up to the end of the line.

3. Chars chart

The chars char lists the characters known by the automaton grouped in a hierachy of classes.

The grammar is ('%' in Spirit means, list of the left part separated the right part) in BNF:

```
start :: classes chars
classes :: "classes" '{' (classdef % ';')? '}'
chars :: "chars" '{' (chardef % ';')? '}'
identifier :: (alnum | '_')+
classdef :: identifier  ('<' identifier)? ':' (char_ - (';'|eol))+
chardef :: hex ',' (char_ - ',')+ ',' identifier (',' >> modifier % ',')?
modifier :: modifiersymbol hex;
```

So, a chars chart is written as

```
classes {
# class definitions
}
chars {
# char definitions
}
```

The spaces class is called `c_b` and it is a kind of delimitor. This is written as

```
c_b < c_del : Blank delimitor ;
```

where `c_del` is an already defined class name. An example of a space character is the tabulation, defined as

```
0009, CHARACTER TABULATION, c_b ;
```

where `0009` is the Unicode code of this characted and `c_b` is its class. But a character can also be associated to some varaiations of itself, introducted by a `modifiersymbol` which is an element in the set {'`m`', '`M`', '`u`'}. For example, 'Ã ' is defined as

```
00E0, LATIN SMALL LETTER A WITH GRAVE, c_a, M00C0, u0061 ;
```

`M00C0` means that its majuscule version is the character with the code 00C0 ('Ã') and `u0061` (u for unmark) means that its unmarked (a generalization of non-accentuated) counterpart is the character with the code 0061 ('a'). The last modifier, unused in this example is 'm' for the minuscule version of majuscule characters.

4. Tokenizer automaton

The automaton is defined by a set of states, each one defined by a set of transitions. When arriving on a state, each transition of the state is tried in order. The first matching transition is used. So the automaton is deterministic. For each transition, The current character in the text is checked against the *event* (a char class) of the rule. If it matches, the (optional) conditions, again char classes, on previous or next characters are checked. If they matches also, the action of the rule is performed. This action can be

- '/' : forget the currently stored chars
- '=' : created a token with the currently stored chars and the current tokenization statuses
- '>' : do nothing

Then, AFTER the action, the (matched with the rule event) current char is added to the currently stored chars and the automaton moves to the target state of the transition, eventually setting the tokenization status of the automaton.

This behavior is defined by a grammar that can be described in BNF-like as:

```
start :: state*
state :: '(' identifier ')' '{' transition* '}'
transition :: '-' precondition* event postcondition* transitionsymbol
identifier ('(' identifier % ',' ')')?
identifier :: (alnum | '_')+
precondition :: '[' (identifier % '|') ']'
event :: identifier % '|'
postcondition :: identifier % '|'
```

The events, pre and postconditions identifiers can be a single char class name of a disjonction of such classes that express that any of the element will match. The pre and post conditions are classes that must appear before or after the current class. The transition symbol is one of the symbols listed above and the comma separated identifiers after the target state identifier are tokenization status defined in the C++ code. These will be defined in the format in a future version.

Automaton have states for the different kinds of tokens (all minuscule, for example) and three special states:

- START : the start state of the automaton which is the state in which is the automaton at the beginning;
- IGNORE : which is reached when reading a blank (spaces, tabulations, etc.) and which will systematically ignore its current char;
- DELIMITER : which is reached when reading a separator like dot, comma, etc. and which will systematically create a token when reading its current char.

5. Development state

The flat tokenizer is up-to-work. Automatons and mm-lp-*.xml files have been converted for all languages but validated (on blocking tva tests) only for eng and fre.