P2X - Universal parser with XML output

Johannes Willkomm

November 7, 2016

Contents

1 Introduction

P2X is a parser, configurable by shortcut grammars, with XML output. The parser uses recursive descent parsing to read any kind of text such as program code, configuration files or even natural language. The input is structured as a tree according to grammar rules which can be specified in a very concise and simple form. The XML output is in a form that makes unparsing particularly simple. This makes P2X suitable for the integration of non-XML data into XML-based projects and/or for source transformation using XSLT.

Consider the following example, were we define solely the binary operator PLUS and specify XML output in merged mode:

```
echo -n "1+2+3" > in.txt
p2x -m -X -b PLUS -o out.xml in.txt
cat out.xml
```

```
<?xml version="1.0" encoding="utf-8"?>
<code-xml xmlns='http://johannes-willkomm.de/xml/code-xml/' xmlns:c='http://johannes-willkomm
 <ROOT>
  <null/>
  <PLUS>
   <INT><c:t>1</c:t></INT>
   <c:t>+</c:t>
   <INT><c:t>2</c:t></INT>
   <c:t>+</c:t>
   <INT><c:t>3</c:t></INT>
  </PLUS>
 </ROOT>
</code-xml>
```

The same output structure can also be represented in MATLAB:

p2x –m –M –b PLUS **in**.txt

```
struct('n','rt','i','','c',{...
 struct('n',{'','op'}, 't',{'','+'}, 'c',{[],...
  struct('n','num', 't',{'1','2','3'})})});
```

or JSON:

p2x –m –J –b PLUS **in**.txt

```
{"n":"rt","i":"","c":[
 {"n":["","op"], "t":["","+"], "c":[{},
  {"n":"num", "t":["1","2","3"]}]}]}
```

2 Usage

P2X works as a stream filter, it reads from standard input and writes to standard output. For example, to read input from file `in.txt` and write the XML file `out.xml`, invoke P2X as follows

```
p2x < in.txt > out.xml
```

P2X can also read from a file given as the first command line argument

```
p2x in.txt > out.xml
```

or write its output to a file given by command line option `-o`

```
p2x in.txt -o out.xml
```

2.1 Program Configuration

P2X can be configured either by options on the command line or from a configuration file, usally `~/.p2x/p2x-options`.

2.1.1 Options

Each option has a long name and some also have a short name. The most important options are the following:

-o, –output specify name of output file

-p, –prec-list specify name of language definition file

-m, –merged set merged output mode

-X, -M, or -J set XML, MATLAB, or JSON output mode

-S specify scanner (lexer)

-g add debug information such as line numbers

The configuration options are listed in entirety in the following table:

Short	Long Option	Description
-h	–help	Print help and exit
	–full-help	Print help, including hidden options, and exit
	–version	Print version and exit
-V	–verbose=¡number¿	Control messages by bit mask (default=‘error,warning’)
	–debug	Enable debugging (default=off)
-p	–prec-list=filename	Precedence file list
-i	–ignore=TokenList	Add an item to ignore
-b	–binary=TokenList	Add a binary operator
-r	–right=TokenList	Add a right associative operator
-u	–unary=TokenList	Add a unary operator
	–postfix=TokenList	Add a postfix operator
-I	–item=TokenList	Add an item
-B	–brace=TokenPair	Scope start and end token
-L	–list-token	List token types (default=off)
-T	–list-classes	List token classes (default=off)
-s	–scan-only	Scan only, do not parse (default=off)
-S	–scanner=name	Select scanner class (default=‘strings’)
-e	–input-encoding=Charset	Input encoding (default=‘utf-8’)
	–stdin-tty	Read from stdin, even if it is a TTY (default=off)
-o	–outfile=Filename	Write output to file `Filename`
	–indent	Indent (default=on)
	–indent-unit=String	Indentation unit (default=‘ ’)
	–newline-as-br	Emit newline text as ca:br element of ca:text (default=on)
	–newline-as-entity	Emit newline text as
character entity (default=off)
-m	–merged	Merge same operator chains, tree will not be binary (default=off)
-w	–sparse	Safe some non-essential attributes, newlines and indents (default=off)
	–write-xml-declaration	Emit XML declaration (with encoding) (default=off)
	–write-bom	Emit byte order mark (BOM) character (default=off)
-O	–output-mode=Mode	Write output as normal (x) or alternative (y) XML, or (J)SON or (M)AT
-M	–matlab	Write output as MATLAB (default=off)
-J	–json	Write output as JSON (default=off)
-X	–xml	Write output as XML (default=off)
	–write-recursive	Recursive output writing (default=off)
-g	–src-info	Emit source location attributes line, column, and character (default=off)
	–attribute-line	Emit attribute line with source line (default=on)
	–attribute-column	Emit attribute column with source column (default=on)
	–attribute-char	Emit attribute column with source char (default=off)
	–attribute-precedence	Emit attribute precedence with token precedence (default=off)
	–attribute-mode	Emit attribute mode with token mode (default=off)
	–attribute-type	Emit attribute type with token type (default=on)
	–attribute-id	Emit attribute id with token id (default=off)

2.1.2 Configuration file

The same options (see previous Section `options`) as on the command line can also be given in a configuration file, which by default searched as `~/.p2x/p2x-options`. In that file there

may be one option per line, short or long, but without the leading `-` or `--`. For example, the following three lines all enable the verbosity level `debug`:

```
V debug
verbose debug
verbose=debug
```

See also environment variable P2X_USER_DIR in Section 2.1.3.

2.1.3 Environment variables

- HOME: Linux only: determine home directory of current user
- HOMEDRIVE, HOMEPATH, APPDATA: Windows only: determine home directory of current user
- P2X_USER_DIR: Set the directory where the configuration file `p2x-options` is found. Default is $HOME/.p2x
- P2X_CONFIG_DIR: If set, this directory is also searched for language definition files
- P2X_DEBUG_INIT: set to non-empty string to turn on debugging of the initialization of P2X

2.2 Language definition

The equivalent of a grammar is called language definition in P2X. It consists of an assignment of token to token classes. Token are those listed by the option `--list-token` or `-L`. Identifiers play a special role, that is, while IDENTIFIER is listed as a single token type, each identifier such as `x`, `name` or `begin` may individually be assigned to a token class. For example `name` might be assigned to class Unary and `begin` might be assigned to class Parentheses.

The token ROOT and JUXTA are special: ROOT is always in class Unary with precedence 0 and JUXTA is always in class Binary, and the precedence and associativity of this operator may be set by the user. Both ROOT and JUXTA do not represent any input.

The available token classes are the following:

- Item
- Binary
- Unary
- Binary_unary
- Postfix
- Ignore

Also, a pair of token may be declared as parentheses. This pair of token encapsulates some subexpression and will be inserted into the parse tree as a whole. The content of a parentheses is itself a parse tree.

A parentheses element can also be assigned to a token class. For example, the parentheses `(` and `)` may be declared as a postfix operator.

The effect of the configuration can be inspected by using the option `-T`.

```
p2x -T -p examples/configs/cfuncs
```

2.2.1 Class Item

All token types, except JUXTA and ROOT, are by default in class Item. Items are represented as tree nodes without children, also called leafs. Two consecutive items are joined automatically by artificial binary JUXTA nodes.

2.2.2 Class Binary

Token types in class Binary represent binary operators, which become binary nodes in the tree, that is, they will usually have two children. A Binary token type has the associated fields associativiy and precedence.

The class Binary is typically used for mathematical binary operators such as +, -, *, / etc. One might also declare identifiers such as `and` or `or` as Binary. Binary token play a crucial role in structuring the tree. For example, to parse a text file into the individual lines, put token NEWLINE in class Binary, presumably with a rather low precedence.

A token from class Binary has a precedence, which is a positive integer, to order the binding strength of two adjacent operators. The typical case is plus + and multiply *, where * has the higher precedence. This way a+b*c is the same as a+(b*c) and a*b+c is the same as (a*b)+c.

A token from class Binary also has an associativity which is either left or right to order two adjacent operators of the same precedence. Usually, arithmetic operators like + and * are left-associative, which means that a*b*c is the same as (a*b)*c. An example for a right-associative operator is the assignment = in C, where x=y=z is the same as x=(y=z).

A token is added to class Binary by one of two ways: first, by using the option `-b` or `--binary`, where you can use token names or identifiers. The token will be given increasing precedences in the order they appear in the command line, from the left to the right, beginning with 1000:

```
p2x -b NEWLINE -b PLUS -bMULT,DIV -btimes,divide
```

The other way is by a `binary` declaration in the laguage definition file, which has the form `token binary` followed by one to four further fields: two integer precedences, an associativity (`left` or `right`) and an output mode (`nested` or `merged`).

```
newline binary 1000 merged
"="     binary 1001 right nested
"*"     binary 1002 merged left
```

2.2.3 Class Unary

Token types in class Unary represent unary prefix operators, which become unary nodes in the tree, that is, they will at most have a single child. The child will always be the right child, that is, the left pointer is always null. A Unary token type has the associated field precedence.

The class Unary is typically used for mathematical unary operators such as +, -. One might also declare identifiers such as `sin` or `cos` as Unary. Another use might be for a line comment delimiter such as `#`.

A token is added to class Unary by one of two ways: first, by using the option `-u` or `--unary`, where you can use token names or identifiers. The token will be given increasing precedences in the order they appear in the command line, from the left to the right, beginning with 2000:

```
p2x  -u 'sin,cos'
```

The other way is by a `unary` declaration in the laguage definition file, which has the form `token unary` followed by an integer precedence field.The command line settings above are equivalent to the following entries in the language definition file:

```
"sin" unary 2000
"cos" unary 2001
```

2.2.4 Class Unary_Binary

Class Unary_Binary is for token which may either occur as unary operators or as binary operators. A typical example are plus `+` and minus `-` in mathematical notation. For that, token may be assigned to class Unary_Binary, which accepts two integer token, the first of which is the binary precedence and the second the unary precedence. An associativity may also be specified. For example, the following configuration

```
"-" unary_binary 1000 2200 left
```

2.2.5 Class Ignore

A token in class Ignore is not used in the construction of the parse tree. However it is not simply discarded. Instead, it is inserted into the tree at the next best convenient location. Thus the text that constituted it is not lost but may be found in the tree as a sort of side information.

In the language definition file, a token is assigned to class Ignore with the keyword `ignore`, but a precedence is also required (it is ignored):

```
NEWLINE ignore 1x
```

2.2.6 Parentheses

A pair of token which is declared as Parentheses encapsulates a subexpression, such that on the outside it appears as a single item, and in fact it is handled by default in the same way as a token from class Item. A Parenthesis definition has a closing list, which is a list of token that may close the subexpression. For example, one could declare the left parenthesis token `(` and the right parenthesis `)` as being a Parenthesis. For each opening token, the Parenthesis declarations are merged together, merging the closing lists. For example, one might declare `while` and `end` and `while` and `endwhile` as Parentheses, then there will be internally just one Parenthesis definition for `while` with the closing list set to the tupel (`end`, `endwhile`).

A token may be added to class Parentheses using the command line option `-B` or `--brace`, which requires as an argument a pair of token separated by ˜,˜ or `:`., for example like this:

```
p2x -B '(:)' -B 'while:end' -B 'while:endwhile' -
```

or using a `paren` declaration in the language definition file:

```
"(" paren ")"
"while" paren "end"
"while" paren "endwhile"
```

A parenthesis expression is be default handled as being in token class Item. However, parentheses declarations may also place assign the declaration to any other class. For example, the parentheses `(` and `)` may be declared as being a postfix operator. This is done by adding the class keyword on the line together with an integer specifying the precedence:

```
"(" paren ")" postfix 10000
```

2.2.7 Example

As an example consider the configuration file `examples/configs/cfuncs` and the input text file `examples/in/cexpr.exp`:

cat examples/configs/cfuncs

```
#
# Operator definitions
#
# Token  Class         Precedence  Unary Prec.  Associativity
"="      binary        5                        right

"+"      unary_binary  10          110
"-"      unary_binary  10          110

"*"      binary        20

"/"      binary        30
"%"      binary        30

" "      ignore        1
NEWLINE  ignore        1
TAB      ignore        1

#
# Parentheses definitions
#
# Start  "paren" End  Class    Precedence  Unary Prec.  Associativity
"("      paren   ")"  postfix  101
```

cat examples/**in**/cexpr.exp

```
f( a +  1)*2
```

```
p2x -mX -p examples/configs/cfuncs examples/in/cexpr.exp
```

```
<?xml version="1.0" encoding="utf-8"?>
<code-xml xmlns='http://johannes-willkomm.de/xml/code-xml/' xmlns:c='http://johannes-willkomm
 <ROOT>
  <null/>
  <MULT>
   <L_PAREN>
    <ID><c:t>f</c:t></ID>
    <c:t>(</c:t>
    <ci:SPACE> </ci:SPACE>
    <R_PAREN>
     <PLUS>
      <ID>
       <c:t>a</c:t>
       <ci:SPACE> </ci:SPACE>
      </ID>
      <c:t>+</c:t>
      <ci:TAB></ci:TAB>
      <INT><c:t>1</c:t></INT>
     </PLUS>
     <c:t>)</c:t>
    </R_PAREN>
   </L_PAREN>
   <c:t>*</c:t>
   <INT><c:t>2</c:t></INT>
  </MULT>
 </ROOT>
</code-xml>
```

3 About this document