P2X - Universal parser with XML output

Johannes Willkomm

October 10, 2018

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"?>
<!-- P2X version 0.6.4 (72b96eb2) -->
<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>
```

One important property of the output is that all input token are present, and that they are present strictly in the order of the input. Thus the entire input text is present in the XML, structured by XML tags. This is even true when some token are set to be ignored. The ignored token are inserted into the tree by attaching them to the next best tree node, but otherwise they have no effect on the shape of the tree.

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

```
<?xml version="1.0" encoding="utf-8"?>
<!-- P2X version 0.6.4 (72b96eb2) -->
<code-xml xmlns='http://johannes-willkomm.de/xml/code-xml/' xmlns:c='http://johannes-willkomm
 <ROOT>
  <null/>
  <PLUS>
   <INT>
    <c:t>1</c:t>
    <ci:SPACE> </ci:SPACE>
   </INT>
   <c:t>+</c:t>
   <ci:SPACE> </ci:SPACE>
   <INT>
    <c:t>2</c:t>
    <ci:SPACE> </ci:SPACE>
   </INT>
   <c:t>+</c:t>
   <ci:SPACE> </ci:SPACE>
   <INT><c:t>3</c:t></INT>
  </PLUS>
 </ROOT>
</code-xml>
```

Any token in the tree has exactly one special child element that contains the input token text, in the example called `c:t`. Ignored token are output as elements with namespace prefix `ci`, in the example called `ci:SPACE`.

Since the token text and ignored elements are in separate namespaces, they do not affect the structure of the tree made up of the elements of the code namespace. Still, the input can be recovered exactly by concatenating the token text and ignored elements in document order. This is done for example by the XSL stylesheet `src/xsl/reproduce.xsl`:

```
xsltproc src/xsl/reproduce.xsl out.xml
```

```
1 + 2 + 3
```

The parsed tree structure can also be represented in MATLAB output:

```
p2x -M -i SPACE -b PLUS in.txt
```

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

or JSON:

```
p2x -J -i SPACE -b PLUS in.txt
```

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

In both MATLAB and JSON format it is not possible to represent all ignored items when using the merged output mode.

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
```

There are currently four different output modes:

- with option `-X` or `--xml` the tree is output in XML format
- with option `-M` or `--matlab` the tree is output in MATLAB format, as a single large `struct` expression
- with option `-J` or `--json` the tree is output in JSON format
- the default is to output an older version of the XML format, which is more verbose

In XML the output tree is as described in this document. In MATLAB and JSON each node has the three fields `n`, `t` and `c`, for the node name, the node text and the node children, respectively.

2.1 Program Configuration

P2X can be configured either by options on the command line or from a configuration file, usually `$HOME/.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)
-n	–noignore	Skip ignored token (default=off)
-l	–loose	Write null elements more loosely (default=off)
	–strict	Write null elements more strictly (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 2.1.1) 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 Environment variables.

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`.

Which token are returned for a given character input depends on the scanner selected with option `--scanner` or `-S`. Since the scanners are defined with Flex there currently is only limited set of scanners to chose from: `c`, `r`, `strings`, `no-strings`. For example, the `no-strings` scanner reports each single quote ~'~ as a token APOS and each double quote ~"~ character as a token QUOTE, and it will never return a token STRING.

The available token classes are the following:

- Item
- Binary
- Unary
- Binary_unary
- Postfix
- Ignore
- Line_comment
- Block_comment

Identifiers play a special role, that is, while IDENTIFIER is listed as one of the token types, each identifier such as `x`, `sin` or `times` may individually be assigned to a token class. For example `sin` might be assigned to class Unary and `times` might be assigned to class Binary. The set of legal identifiers also depends on the selected scanner, but as of now all scanners have the same definition.

The basic definition of an identifier is based on the C model. Identifiers may also contain any of the upper Unicode characters. Currently the hyphen character may not occur in an identifier, for example, "a-b" will be scanned as three tokens.

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 ROOT token is always present as the root of the parse tree. The JUXTA token is automatically inserted in the parse tree as a binary operator whenever two consecutive items are encountered.

By default all other token are in class Item.

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-special/cfuncs.p2c
```

Generally token can either by referred to by the name of the token class as listed by the option `-L` or they can be referred to literally, in quotes. For example, the following two lines are equivalent:

```
MULT    binary 1002
"*"     binary 1002
```

Currently there is no way to see the regular expression used by the scanner for each token type, and no way to see which set of input words is represented by a given token. Also there is currently no documentation telling you that the token returned by the scanner for the input `*` is called MULT. Hence it is probably preferably to use the second form. For individual identifiers the only option is of course to use the quoted form.

Example configuration files can be found in the directory `examples/configs`. The file `./examples/configs/default` is used by the P2X test suite and showcases all available declarations and options.

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
```

In the output tree the binary operators are represented as binary nodes with two children, their left and right operands. This can lead to deeply nested trees. To mitigate this P2X can also output the binary nodes in *merged* mode. Then n consecutive operators made of the same token are collapsed into a single node with $n+1$ children. The option `-m` or `--merged` activates the merged output globally and the `merged` keyword in a language definition declaration can activate merged mode for individual operators in class Binary.

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
```

In the output tree Unary operators are output as a node with a two child nodes, as they can be seen as a binary operator with the left operand missing. The missing left operand is marked with a special `null` element, so that the single operand is the second child element.

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 options, 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
```

The same input `-3` can now result in either a unary node or a binary node in the output tree, depending on the preceding token. When an Item is pending because the preceding token was of class Unary or Binary the minus `-` will result in a unary operator node, otherwise as a binary node.

2.2.5 Class Postfix

The class Postfix is complementary of the Unary class. It defines a unary postfix operator. Typical examples are the operators ˜'˜ and ˜.'˜ in MATLAB. Another example would be to define units as postfix operators:

```
"kg" postfix 10000
"m" postfix 10000
"s" postfix 10000
```

Another important application are the typical function call and array access expressions in may languages, such as `f(x)` or `x[2]`. These are most conveniently treated as postfix operators, which is possible with P2X because each pair of parentheses can also be given a token class, see section 2.2.7.

```
"(" PAREN ")" postfix 10000
"[" PAREN "]"  postfix 10000
```

In the output tree Postfix operators are output as a node with a single child, as they can be seen as a binary operator with the right operand missing.

2.2.6 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 1
```

In the XML output token in class Ignore are represented using a separate namespace, so for many intents and purposes they are as good as invisible. However, for input reconstruction it is of course crucial that they are kept.

Keeping the Ignore token one can reconstruct input identically even if its structure is unknown. P2X guarantees that all input characters, including those from token in class Ignore also occur in the output XML document, in the same order as they occured in the input.

In the MATLAB and JSON output formats it is not possible to represent the token in class Ignore, so they are truly ignored and discared then.

2.2.7 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 be assigned to any other token class. This can be done for each parentheses declaration individually. For example, the parentheses `(` and `)` may be declared as being a postfix operator, to represent the typical function call expression like `f(x)`. This is done by adding the class keyword on the line together with an integer specifying the precedence:

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

Another way to parse a function call expression is to leave the parentheses definition in token class Item and rely on the JUXTA operator which will be inserted between the function name – provided that is parsed as an Item – and the parentheses. One will probably give a rather high precedence to the JUXTA operator then.

```
"(" paren ")"
JUXTA binary 2000
```

2.2.8 Line comments and block comments

Line comments and block comments can be used to escape from the normal parsing parse comment text sections. Line comments start with a certain token and continue until the next NEWLINE token. Block comments work like parentheses and have start and end tokens.

All the text belonging to a comment is collected and inserted as a single Ignore item into the parse tree. In the case of the line comment the terminating NEWLINE is inserted individually, according to the rules defined for it.

For example, the two modes of comments used in the C++ language can be configured in P2X as follows:

```
"//" line_comment 1
"/*" block_comment "*/"
```

2.2.9 Ignoring tokens depending on context

There are two special rules that can override the normally configured mode for certain token classes depending on the context. The first is the flag `ignoreIfStray` that can be used with Binary token. The effect is that a token from that class is treated as an Ignore token when there is an open operator (Unary or Binary) in the tree already.

The other special rule is `A ignoreAfter B`, which means that a token A is treated as Ignore whenever it occurs after a B token.

For example, a special token to escape a NEWLINE, as in MATLAB, can be configured in P2X as follows:

```
NEWLINE    BINARY 1000
"..."      IGNORE 1
NEWLINE    ignoreAfter "..."
```

2.2.10 Example

As an example consider the configuration file `examples/configs-special/cfuncs.p2c` and the input text file `examples/in/cexpr.exp`:

cat examples/configs-special/cfuncs.p2c

```
#
# Operator definitions
#
# Token  Class         Precedence  Unary Prec.  Associativity  Options
","      binary        3                                      merged
```

```
";"      binary        3                                        merged
"return" unary         4
JUXTA    binary        5
"="      binary        6                        right
"=="     binary        7

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

"*"      unary_binary  20          120

"/"      binary        30
"%"      binary        30

"if"     unary        100
"while"  unary        100

#
# Parentheses definitions
#
# Start  "paren" End  Class    Precedence  Unary Prec.  Associativity
"("      paren   ")"  postfix  101
"["      paren   "]"  postfix  102
"{"      paren   "}"

" "      ignore        1
NEWLINE  ignore        1
TAB      ignore        1
```

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

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

p2x -S c -lnX -p examples/configs-special/cfuncs.p2c examples/**in**/cexpr.exp

```
<?xml version="1.0" encoding="utf-8"?>
<!-- P2X version 0.6.4 (72b96eb2) -->
<code-xml xmlns='http://johannes-willkomm.de/xml/code-xml/' xmlns:c='http://johannes-willkomm
 <ROOT>
  <MULT>
   <L_PAREN>
    <ID><c:t>f</c:t></ID>
    <c:t>(</c:t>
    <R_PAREN>
     <PLUS>
      <ID><c:t>a</c:t></ID>
```

```
      <c:t>+</c:t>
      <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>
```

As a more complex example try parsing an entire program:

```
p2x -lnX -S c -p examples/configs-special/cfuncs.p2c examples/in/cfunc.exp
```

3 About this document