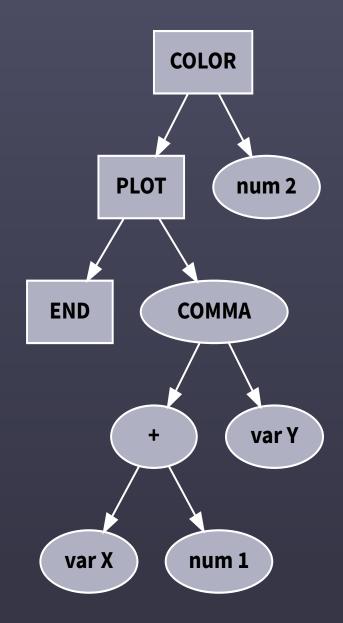# basicParser

Turbo-Basic XL and Atari BASIC parser tool

https://github.com/dmsc/tbxl-parser

Turbo-Basic XL and Atari BASIC parser tool

This program parses and tokenizes a *Turbo-Basic XL* or *Atari BASIC* listing in a flexible format and produces any of three outputs:

- A tokenized binary file, directly loadable in the original *Turbo-Basic XL* (or *Atari BASIC* if the `-A` option is given) interpreter. This mode also replaces variables with single letters by default, but with the `-f` option writes the full variable names and with the `-x` option writes empty variable names, making the program unable to be listed or edited.

  This is the default operating mode, and also can be forced with the `-b` command line switch.

- A minimized listing, replacing variable names with single letters, using abbreviations, removing spaces and using Atari end of lines.

  This mode is selected with the `-s` command line switch. Adding the `-f` option keeps the names of variables with 2 or less characters.

- A pretty printed expanded listing, with one statement per line and indentation, and standard ASCII line endings.

  Note that this format can be read back again, but some statements are transformed in the process, this can lead to problems in non-standard `IF` / `THEN` constructs.

  Currently, `IF` / `THEN` with statements after the `THEN` are converted to multi-line `IF` / `ENDIF` statements.

  This mode is selected with the `-l` command line switch.

The input listing format is very flexible:

- Standard Atari listing format, with Atari or ASCII end of lines. The parser understand the same abbreviations available in the original interpreter.

- A free format listing, with line numbers separated from the statements.

  In this format, not each line needs line number, only lines that are target to `GOTO` / `GOSUB` / `THEN` needs them. If you use only labels, no line numbers are needed.

  Also, line numbers can appear alone in a line, for better readability.

- Adds parameters and local variables to `PROC` and `EXEC` .

  Arguments follow the `PROC` label after a comma, and local variables follow after a semicolon:

```
1    D = 3
2    EXEC Testing, D+5, "Hello"
3    PRINT D
4    PROC Testing, A, B$(10); D
5        D = A + 1
6        PRINT D; " and "; B$
7    ENDPROC
```

  As the example shows, string variables must include the dimensioned length, as the parser adds a `DIM` at the start of the program to initialize. The dimensioned length must be an integer, a `$define` or a `%` number.

  Also, setting the value of variable "D" inside the procedure does not alter the value of the variable "D" outside the procedure.

  The parser transform this construct by creating new variables that hold the parameters and local variables, so the resulting procedures don't support recursion.

- Inside strings, special characters can be specified by using a backslash followed by an hexadecimal number in upper-case, (i.e., `"\00\A0"` produces a string with a "hearth" and an inverse space), this allows editing special characters on any standard editor. Note that to force a backslash before a valid hex number, you can use two backslashes (i.e., `"123\\456"` produces `123\456`).

- Comments can be started by `'` in addition to the *Turbo-Basic XL* `.` , `--` or `rem` .

- The input is case insensitive (uppercase, lowercase and mixed case is supported).

- There are parsing *directives* added, that consist on lines starting with a dollar sign `$` . A list of available directives documented bellow.

- There is support for extended strings, with embedded character names. Start the string with `["` and end the string with `"]` , and include special characters with `{name}` or `{count*name}` , with count a decimal number and name from the list: `heart` , `rbranch` , `rline` , `tlcorner` , `lbranch` , `blcorner` , `udiag` , `ddiag` , `rtriangle` , `brblock` , `ltriangle` , `trblock` , `tlblock` , `tline` , `bline` , `blblock` , `clubs` , `brcorner` , `hline` , `cross` , `ball` , `bbar` , `lline` , `bbranch` , `tbranch` , `lbar` , `trcorner` , `esc` , `up` , `down` , `left` , `right` , `diamond` , `spade` , `vline` , `clr` , `del` , `ins` , `tbar` , `rbar` , `eol` , `bell` .

Example Programs

There are two sample programs, located in the `samples` folder that illustrate the free-form input format.

```
1    ' Example program
2
3    ' One statement per line:
4    print "Hello All"
5    print "--------"
6    print "This is a heart: \00"
7
8    ' Also, multiple statements per line:
9    for counter = 0 to 10 : ? "Iter: "; counter : next counter
10
11   ' Line numbers
12   30
13   ' And abbreviations:
14   g. 30
```

To generate a tokenized BAS file, loadable by *Turbo-Basic XL*, simply type:

```
1    basicParser samples/sample-1.txt
```

This will generate a `sample-1.bas` file in the same folder.

If on the other hand you want a minimized listing file in ATASCII format (suitable for `ENTER` into *Atari BASIC*, type:

```
1    basicParser -l -A samples/sample-1.txt
```

This will generate a `sample-1.lst` file in the same folder.

Program Usage

```
1    basicParser [options] [-o output] filenames
```

Options:

- `-n nun` Sets the maximum line length before splitting lines to `num`. Note that if a single statement is longer than this, the line is output anyway. The default is 120 characters (the standard Atari Editor limit)

- `-l` Output long (readable) listing, suitable for editing, with standard end of lines and lowercase statements.

- `-s` Output a short, minimized listing, with ATASCII end of lines. The default output file name is the same as input with `.lst` extension added.

- `-b` Output a binary tokenized file instead of a listing. The default output file name is the same as input with `.bas` extension added. Note that this is the default behaviour.

- "-A" Accept (and produce) standard *Atari BASIC* language, without the extended statements and syntax. Note that some of the optimizations are specific to *Turbo-Basic XL* and won't run in this mode.

- `-x` In binary output mode, writes null variable names, making the program unlistable. This options does nothing on listing output.

- `-f` In binary output mode, writes the full variable names, this eases debugging the program. In short listing mode, keeps the names of variables with less than two characters, renaming all longer or invalid names.

- `-k` In binary output mode, keeps comments in the output. Note that only standard comments are included, not new style (`'`) comments.

- `-a` In long output, replace Atari characters in comments with approximating characters.

- `-v` Shows more parsing information, like name of renamed variables. (verbose mode)

- `-q` Don't show any parsing output, only errors. (quiet mode)

- `-o` Sets the output file name. By default, the output is the name of the input with `.lst` (listing) or `.bas` (tokenized) extension. If the given name starts with a dot, use as output file name extension.

- `-c` Output to standard output instead of a file.

- `-h` Shows help and exit.

Parser directives

Directives add extra features to the parser, much like C and C++. Directives start with a dollar as the first non blank character on a line, and continue up to the end of the line.

Bellow is a description of available directives.

$options directive.

The options directive alter the way the parsing is done, accepting a list of comma separated options, valid for the current file. Valid options:

- `mode=compatible` : Disable features to be more compatible with the *Turbo-Basic XL* parser.
- `mode=extended` : Makes the parser to accept more extended features.
- `mode=`**`default`** : Returns the parser to the default mode.
- `optimize` or `+optimize` : Allows the parser to optimize the output to produce smaller or faster code.
- `-optimize` : Disable the optimizations.
- `optimize=+` *suboption*: Enable the particular optimization option.
- `optimize=-` *suboption*: Disable the particular optimization option.

The optimization sub-options are:

- `const_folding` : Replace operations on constants with the result.
- `convert_percent` : Replace small integers with the `%*` equivalent, this is only available in *Turbo-Basic XL* mode.
- `commute` : Swap arguments to binary operations to minimize runtime.
- `line_numbers` : Remove all Basic line numbers that are unused.
- `const_replace` : Replace repeated constant values (numeric or string) with a variable initialized to the value. The initialization code is added before any statement in the program, and tries to use the minimum number of bytes posible.

Note that options can be changed at any place in the file, this is an example of changing the parser mode in the middle of the file:

```
1   ' Example program using directives
2   $ options optimize, mode=default
3   error1 = 2
4   ? error1   : ' This is parsed like Turbo-Basic XL, as ? ERR OR 1
5
6   $options mode = extended
7   ? error1   : ' This is parsed as ? error1
8   Printa     : ' This is a parsing error.
```

A good optimization mode for producing short listings is:

```
1   $options +optimize, optimize=-convert_percent-const_replace
```

The above line instructs the parser to avoid converting numbers to `%` values and the replacement of constants, producing a smaller listing. Note that replacement of constants can be beneficial, so try enabling the optimization and running with "-v" option to see what variables are good candidates for replacement.

`$define` directive.

This directive defines new symbols that are replaced at parsing time with the values, like C macros.

Replacement names are prefixed by `@` to differentiate from variables, and as variables, string defines end in `$`, the syntax of the directive is:

`$define` *defineName* `=` *value*

Keep in mind that as the value is replaced each time the variable is used, it is probably best to assign them to a variable instead if the value will be used multiple times, and you should enable optimizations so that the usage is simplified at parsing time.

This is an example usage of the `$define` directive:

```
1    ' Example usage of defines
2    $options +optimize
3    $define Message$ = "Hello world!"
4    $define PCOLR0   = $2C0
5
6    print @Message$        : ' Replaced by: ? "Hello world!"
7    print len(@Message$)   : ' Replaced by: ? 12
8    poke @PCOLR0+2, $1F    : ' Replaced by: POKE 706,31
```

`$incbin` directive.

This directive allows including data from a binary file to a new string definition. The content of the file is read at parsing time and the full content is stored in the define. The syntax of the directive is:

`$incbin` *defineName$* `,` `"` *fileName* `"` [`,` *offset* [`,` *length*]]

The optional *offset* parameter specifies a starting offset in bytes for the included data, and the optional *length* parameter specifies the number of bytes to read. If *length* is not given, the file read completely.

This is an example usage of the `$incbin` directive:

```
1    $options +optimize
2    $incbin asmBin$, "myasm.bin"
3
4    asmRut = adr( @asmBin$ ) : ' Store address in variable to use
        multiple times.
5    ? usr(asmRut, 1, 2)      : ' Call routine. Should be relocatable and
        less than 242 bytes.
```

Limitations and Incompatibilities

There are some incompatibilities in the way the source is interpreted with the standard *Turbo-Basic XL* and *Atari BASIC* parsers:

- The ASCII LF character (hexadecimal $10) is interpreted as end of line in addition to the ATASCI EOL (hexadecimal $9B). This means that in `DATA` statements and comments the LF character is not accepted.

- The parsing of special characters inside strings means that a valid hexadecimal sequence (`\**` , with `*` an hexadecimal number in uppercase) or two backslashes are interpreted differently.

- Extra statements after an `IF` / `THEN` / `LineNumber` are converted to a comment, with the exception of `DATA` statements. In the original, those statements are never executed, so this is not a problem with proper code.

- Any string is accepted as a variable name, even if it is already an statement, function name or operator.

  The following code is valid:

  ```
  1    PRINTED = 0      : ' Invalid in Atari BASIC, as starts with "
           PRINT"
  2    DONE = 3         : ' Invalid in Turbo-Basic XL, as starts with "
           DO"
  ```

  This relaxed handling of variable naming creates an incompatibility, as the first example above is parsed differently as the standard *Atari BASIC*, where it means " `PRINT (ED = 0)` " instead of " `LET PRINTED = 0` ".

  Note that currently, even full statements are accepted as variable names, but avoid using them as they could produce hard to understand errors.

- In long format listing output, `IF` / `THEN` are converted to `IF` / `ENDIF` statements. This introduces an incompatibility with the following code:

  ```
  1    FOR A = 0 TO 2
  2      ? "A="; A; " - ";
  3      IF A <> 0
  4        ? "1";
  5        IF A = 1 THEN ELSE
  6        ? "2";
  7      ENDIF
  8      ? " -"
  9    NEXT A
  ```

  This code should produce the following at output:

```
1    A=0 - 2 -
2    A=1 - 1 -
3    A=2 - 12 -
```

After conversion, the `ELSE` is associated with the second `IF` instead of the first, giving the wrong result.

- Parsing of `TIME$=` statement allows a space between `TIME$` and the equals sign, but in *Turbo-Basic XL* this gives an error.

Compilation

To compile from source, you need `gawk` and `peg`, both are available in any recent Debian or Ubuntu Linux distro, install with:

```
1  apt-get install gawk peg
```

To compile, simply type `make` in the sources folder, a folder `build` will be created with the executable program inside.