ULS
User Guide

(ver-0.8.1)

Nov 2022

Stanley Hong

link2next@gmail.com

Table of Contents

License: The MIT License

1. Overview

ULS(Unified Lexical Scheme) is the system to create lexical analyzers from lexical specification files. It aims at providing compiler creators with an intuitive, optimized language design of lexical analysis.

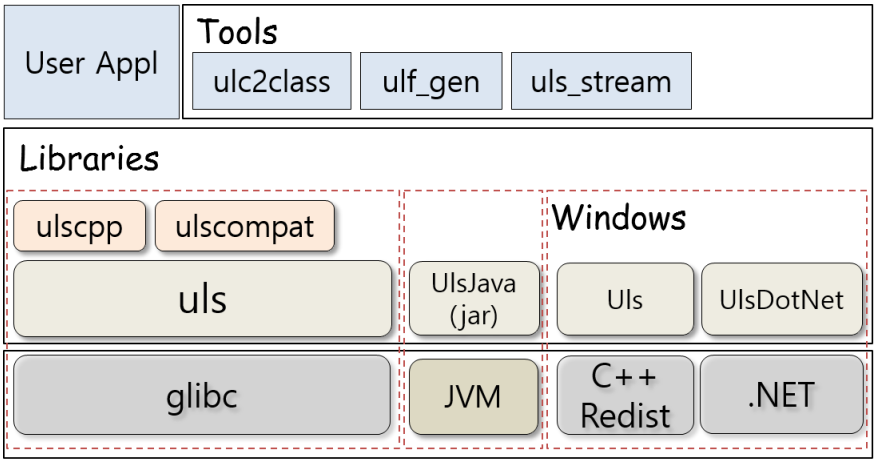
ULS support the languages C/C++, C#, and JAVA. It has been written in C and C++.NET and is built as C-library in Linux and as .NET dll in Windows. It's given as class libraries wrapping the base C or C++.NET library for other languages. The followings are currently the supported language base on which ULS stands.

- ✧ C++: stdc++ in Linux and vs2022 on Windows
- ✧ Java JNI : The class/jar and example source of java is included in the main source tree since the version 1.8.13
- ✧ C#: based on C++.NET dll library

The ULS can be built on 64bit platform and tested on the followings.

- ✧ LinuxMint 20.2
- ✧ Ubuntu 18.04
- ✧ OpenSUSE 12.3
- ✧ Centos 6.8
- ✧ Windows 10

The package consists of core libraries, wrapper libraries, and a few of tools. The core library are uls(so,dll) and UlsDotNet(dll). The executables, ulc2class, ulf_gen, and uls_stream, are utilities for accessing the system. Below is the diagram to show library layers.



The libraries are implemented as shared library(*.so) on Linux and dynamic linked library(*.dll) in Windows. Uls.dll is a compound of the sources of 'uls' and 'uls_compat'. UlsDotNet is for Windows.

2. Installation

2.1. The installation on Linux

2.1.1. Using deb file on Linux

On Ubuntu, Mint, or Debian, a deb-file, `uls-1.8.x-amd64.deb,` are provided.

To install the deb-file on 64-bit arch, download the file on the site. Enter the following command.

```
$ sudo gdebi uls-1.8.x-amd64.deb
```

Now, the package has been installed and try listing the installed files.

```
$ dpkg –L uls
$ which ulc2class
$ ulc2class –h
```

The examples and doumentaion of ULS can be installed by 'setup_up_examples'.

```
$ setup_uls_example ~
$ cd ~/uls_examples
$ make
```

To uninstall the package,

```
$ sudo dpkg –P uls
```

2.1.2. Build ULS from source files on Linux/macOS

Assume you've got the source tarball named `uls-1.8.x.tar.xz` from the site. There is a shell script file called 'configure' in the top directory of source tree when you untar the tar-xz file. Run it as follows:

```
$ ./configure
```

The default installation directory is /usr/local. If you want the other directory, you can set an option in the 'configure' stage:

```
$ ./configure --prefix=<INST_DIR>
```

Then build the binaires by issuing the command 'make':

```
$ make
```

Before installing, you may test the software if the resultant libraries and executables are correctly working on your machine.

```
$ make check
```

With the administration privilege of the system, install the files:

```
$ sudo make install
```

2.2. The installation on Windows

2.2.1. Installation on Windows

There is a self-extracting install program uls-windows-1.8.x-x64.exe in the site. It contains the binary files which can be used on x86_64/amd64. Download and run it. Follow the directions in the setup.

NOTICE: The installer of Uls on Windows is likey to cause the issue, '**False Positive Antivirus Detection**'.

After installation, check if the path for uls utilities has been appended to the environment variable 'PATH'. The value of 'PATH' might be identified in the following path in **'My Computer' -- 'Properties'.**

'Advanced' Tab -- 'Environment Variables' -- 'User variables for...'

Try the following commands to see the usages of uls-tools in dos-prompt.

```
ulc2class -h
uls_stream -h
ulf_gen –h
```

2.2.2. Building ULS from source files on Windows.

The ULS source of Windows is also distributed with VisualStudio™ solution files.

A. After unpacking the source distribution, uls-windows-1.8.x.tar.xz,

   run VisualStudio 2022 to open the solution file vs2022₩UlsWin.sln.

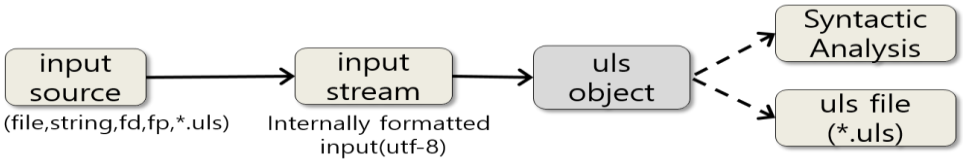
B. Choose the configuration **Release/x64**. Notice only the 'Release' mode is supported currently.

C. Click the 'Build Solution' button. When the build-process is finished, all the executables and libraries will be located in x64/Release in 'vs2022'

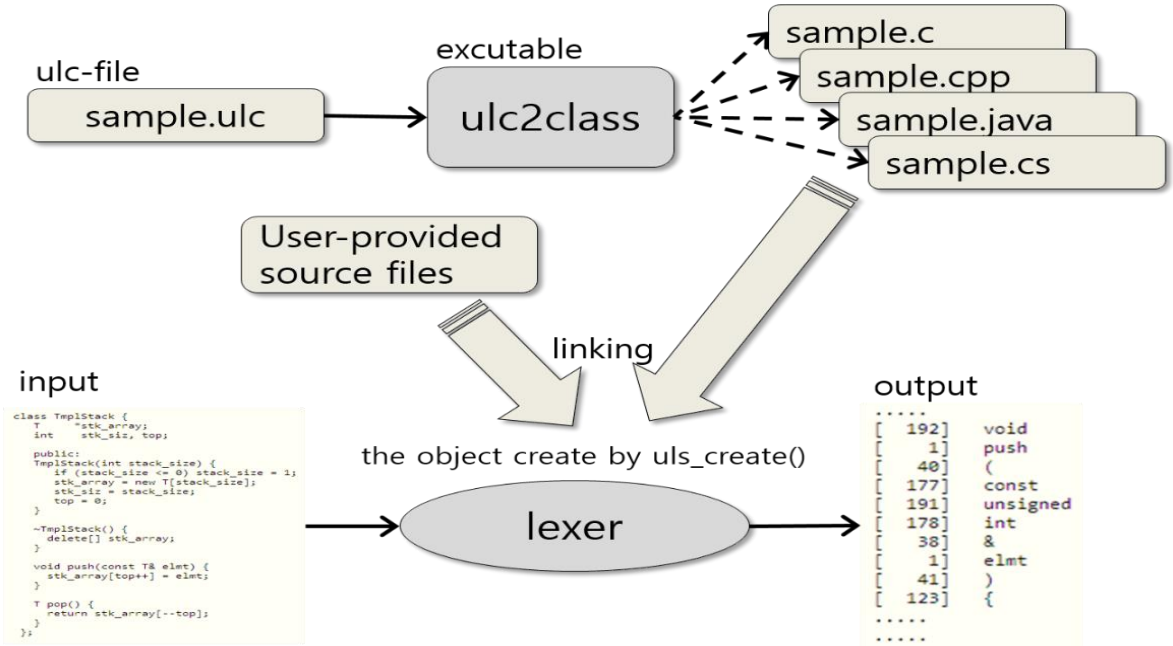
3. The Lexical analyzer

The token in lexical analysis is defined as an integer and its associated information. The major function of lexical analysis is to give user a token sequence from given input so that the resultant token sequence can be the input of the syntactic analysis in the compiling process. The main purpose of the project is to provide the developers of compilers with an easy way to create lexical analyzer as objects.

The input of the lexical analyzer can be from various sources. The 'input source' in ULS is defined as the input of the 'input stream' and can be file, string, binary-format-file(*.uls) which is generated by a uls tool, or arbitrary object implementing (ULS) specific interface. The formats of input file as input-source can be encoded in utf-8. The output format is encoded in utf-8. The data flow in ULS is as follows.



The lexical analyzer object is created by reading the lexical configuration file whose suffix is '.ulc'. The ulc-file is a utf-8(with BOM) encoded text file that describes lexical attributes and token list for a (programming) language. Below is illustrated the brief process that creates the output from programming source files.



1. Create new ulc-file. You can copy and edit the one of the examples (*.ulc) in the shared directory. There'is also a minimal ulc-file named 'simplest.ulc'.
2. Generate the source files from the ulc-file and link it with your source files. The languages of source file can be chosen in C/C++, C#, Java by option of the command **ulc2class**.
3. Create the uls lexical object by calling the ULS-API in your application
4. Prepare input file and start tokenizing using uls (class) API.
5. Get the token sequence from the input file. The pair of token-id(int) and token-string(lexeme) should be the input of the next step of compiling, syntax analysis.

3.1. Key Features

The ULS system has the following features.

- ✧ ULS can generate classes which can be linked with your own source files which might be written in prevalent modern programming languages like C/C++, Java, C#.
- ✧ It can save the tokens from many input files to intermediate file, which can be replayed again. The intermediate file is used much like token streaming file.
- ✧ It can manipulate number of arbitrary length of digits as it stores the numbers as text format,
- ✧ User programs can instantiate multiple objects for lexical analysis as the system is designed with concept of objects in mind

4. ULS Configurations

4.1. ULC File

To create the lexical object for a specific compiler, firstly the users have to determine which word is mapped to keyword. A file for lexical configuration should be written by user and given to ULS so that ULS can create a lexical object. After editing the ulc-file, run 'ulc2class', which generates a proper source code file having class definition.

There're 'reserved' tokens that are initially defined by the ULS system:

- ✧ EOI: The last token of input. It returns EOI if the input reachs the end of input.
- ✧ ERR: It returns this err token in case of an error occurring.
- ✧ ID: Identifer token, which should be also defined by user in ulc-file.
- ✧ NUMBER: It represents the token that is number, integer or floating number.
- ✧ LINENUM: This informs user of the current processing position of the input.
- ✧ TMPL: This allows placeholders in uls-file to be replaced with actual strings.
- ✧ NONE: In case the input stack of the analyzer is at initial state, the current token is NONE.

The ulc-file consists of 2-sections. The second part of ulc-file defines 'regular' tokens.

4.1.1. Lexical Attributes

The ulc-file is composed of 2 sections which are separated by the string '%%'. The first part has a few of options characterizing your own programming language. The second part is the list of token definitions one per line.

```
#@ulc-2.3
# one line comment starts with character '#'
# The first is to describe the lexical attribute of language.

# A separator '%%' between the header and body of the file.
%%

# Here comes a lexical token definition per one line.
INT int 256
FLOAT float
ARROW ->
WORD
```

The lexical attributes in the first section contains multiple of lines upto the separator '%%. Each line represents one (lexical) attribute. The attribute starts with its name trailed by colon character. For

example, 'COMMENT_TYPE:'. You can make use of the following attributes.

- **COMMENT_TYPE:** *start-comment-mark end-comment-mark*

  COMMENT_TYPE specifies a comment-style. To define the familiar C-style comment, the following line should be added to the file.

  ```
  COMMENT_TYPE: /* */
  ```

- **QUOTE_TYPE: [token-specifier] [option-specifier]** *start-mark* [*end-mark*] **[verbatim0 | verbatim1]**

  token-specifier : **token=**[*token-number*],[*token-name*]

  option-specifier : **option=***list-of-options*

  QUOTE_TYPE specifies a literal string style of your language. For the familiar C-style quotation, this can be like this.

  ```
  QUOTE_TYPE: option=asymmetric @" "
  QUOTE_TYPE: token=100,SQUOTE ' verbatim1
  ```

  By default, ULS gives the literal string with escape-chars processed, but if you want to get the unanalyzed(unescaped) literal string, append the keyword 'verbatim1' to the end of the line.

  ```
  QUOTE_TYPE: ' ' verbatim1
  ```

  The mode 'verbatim1' is the mode where escaped of starting char is only processed. In default mode, all legacy characters for escaping are processed:

  ```
  \n      ---> new line
  \r      ---> carrage return
  \t      ---> horizontal tab
  \b      ---> back space
  \a      ---> alert
  \f      ---> form feed
  \v      ---> vertical tab
  \'      ---> single quote
  \"      ---> double quote
  \\      ---> backslash
  \xNN    ---> hexadecimal char
  \NNN    ---> octal number
  ```

  The By default, ULS gives the literal string with escape-chars processed, but if you want to get the unanalyzed(unescaped) literal string, append the keyword 'verbatim1' to the end of the line.

  The token-specifier is to specify the the token number or name assigned to the quote string. The speicfiers

The option-specifier is to specify the option of quote-string:

- ✧ **asymmetric**: You can omit the ***end-mark*** .
- ✧ **multiline**: The quote string spans over multiple lines.

The options should be separated by the commas without any space characters.

♦ **ID_FIRST_CHARS:** *charset1 charset2* **...**

ID_FIRST_CHAR specifies the character set in which character can be the first-char of identifier. ID_FIRST_CHAR by default contains all the alphabet characters, strictly speaking, 'A-Z a-z', as default character set. The character set you specify will be just added to this default character set. You can directly specify the characters without the quotation mark as follows

```
ID_FIRST_CHARS: $ . _
```

Make sure not to attach the characters as follows.

```
ID_FIRST_CHARS: $._
```

♦ **ID_CHARS:** *charset1 charset2* **...**

ID_CHAR specifies the character set that constitutes identifier. The default set of 'id-char-set' consists of alphabets. ID _CHAR, by default, contains all the alphabet characters, strictly speaking, 'A-Z a-z', as default character set. Use the dash character to specify the range. You can also hexa-decimal range as follows.

```
ID_CHARS: 0-9 0x123-456
```

♦ **RENAME: [ID|NUMBER|EOI|EOF|ERR|LINENUM|NONE]** *new-name*

The ULS system internally defines a few of reserved keyword for identifier, number, end-of-input, end-of-file, and so on. Their names are ID, NUMBER, EOI, EOF and the like. If you want to change these reserved names or even the token number, use 'RENAME:' attribute.

```
RENAME: ID IDENT
RENAME: NUMBER NUM 100
```

The default token number of reserved keywords can be shown by ulc2class.

```
$ ulc2class simple
$ cat SimpleLex.h
```

♦ **CASE_SENSITIVE: [true|false]**

The keyword and identifier are case-insensitive in some languages like SQL. Use this attribute.

```
CASE_SENSITIVE: false
```

The default value of the attribute is true.

- **ID_MAX_LENGTH:** *maximum-length*

  This attribute is used to restrict the length of identifier.

  ```
  ID_MAX_LENGTH: 64
  ```

- **NUMBER_SUFFIXES:** *suffix-string* **...**

  This attribute is used to specify the constant suffixes of numbers, integer floating-number, or complex number. Most of the programming languages calls for suffices for number like 'UL', 'ul', 'LU', 'L', 'f' 'i'. For example, you can use the attribute like this

  ```
  NUMBER_SUFFIXES: UL ul LU L f j
  ```

- **DECIMAL_SEPARATOR:** *one-char*

  The separator may be one character. For example, you can specify the separator in ulc-file.

  ```
  DECIMAL_SEPARATOR: _
  ```

- **NUMBER_PREFIXES:** *list-of-pairs*

  To specify the radix of number, you can use attribute NUMBER_PREFIXES in ulc-file.

  ```
  NUMBER_PREFIXES: 0:8 0x16 0b:2 0o:8
  ```

  This is the list of pairs where the item is number prefix and radix

Make sure not to separate the ':' from attribute name in all cases.

4.1.2. Token Definition Line

The second part of the configuration file begins next to the '%%'-line. Each line defines one keyword and consists of three columns, **keyword-name**, **keyword-string**, **token-number**. The **keyword name** is mandatory and the others are optional. In the line without token number, it's derived from the previous line. The line without keyword-string is just the token to be used internally in the lexical analysis.

4.2. How to generate source files

Once the lexical configuration file(*.ulc) is edited, the executable 'ulc2class' can generate C/C++ header-file defining tokens in terms of your lexical specification. The input file of it is the lexical configuration file(*.ulc) that defines the lexical attributes of the language you want to implement.

The information assigned to the tokens is generated with the tool **ulc2class** in this way

```
$ ulc2class simple
$ cat SimpleLex.h

#include <uls/UlsLex.h>

class Simple : public uls::crux::UlsLex {
  public:
       static const int            ERR = -8;
       static const int           NONE = -7;
       static const int           LINK = -6;
       static const int           TMPL = -5;
       static const int         LINENUM = -4;
       static const int          NUMBER = -3;
       static const int              ID = -2;
       static const int             EOF = -1;
       static const int             EOI = 0;
       Simple();
};
```

The default generated language of ulc2class is C++:

$ ulc2class -n AAA.BBB.SampleLex sample.ulc

```
#include <uls/UlsLex.h>

namespace AAA {
    namespace BBB {
        class SampleLex : public uls::crux::UlsLex {
         public:
            enum Token {
                ERR = -1,
                EOI,
                ID,
                NUM,
                ... ...
            };

            SampleLex(std::string& ulc_filepath) : UlsLex(ulc_filepath) {}
            virtual ~SampleLex() {}
        }
    }
}
```

The other language can be generated by –l-option like -lc, -ljava, -lcs. You can specify the long name containing namespaces with -n-option like 'AAA.BBB.SampleLex', or just class name 'SampleLex'.

```
To generate c++ header, which is the default mode.
  $ ulc2class sample.ulc
  $ ulc2class -n AAA.BBB.SampleLex sample.ulc

To generate c header file(*.h), use -l-option.
  $ ulc2class -lc sample.ulc
  $ ulc2class -lc -e sample.ulc
  $ ulc2class -lc -e -n e_name sample.ulc

To generate C# wrapper class file(*.cs),
  $ ulc2class -lcs -n AAA.BBB.SampleLex sample.ulc
  $ ulc2class -o /topdir/AAA/BBB/SampleLex.cs -lcs -n AAA.BBB.SampleLex sample.ulc

To generate java wrapper class file(*.java),
  $ ulc2class -ljava -n AAA.BBB.SampleLex sample.ulc
  $ ulc2class -o /topdir/AAA/BBB/SampleLex.java -ljava -n AAA.BBB.SampleLex sample.ulc
```

To create header file for c source,

```
 $ ulc2class –pTOK_ -lc sample.ulc
 $ cat sample_lex.h
 #define TOK_ERR     -1
 #define TOK_EOI      0
 #define TOK_EOF      1
 #define TOK_ID       2
 #define TOK_NUMBER   3
 ... ...
```

As shown above, the token number is assigned to the macro of the token and the token numbers can be any of integer including negative numbers only if there are no conflictions with the reserved tokens. The –p-options can be used to prepend 'TOK_' to each token name.

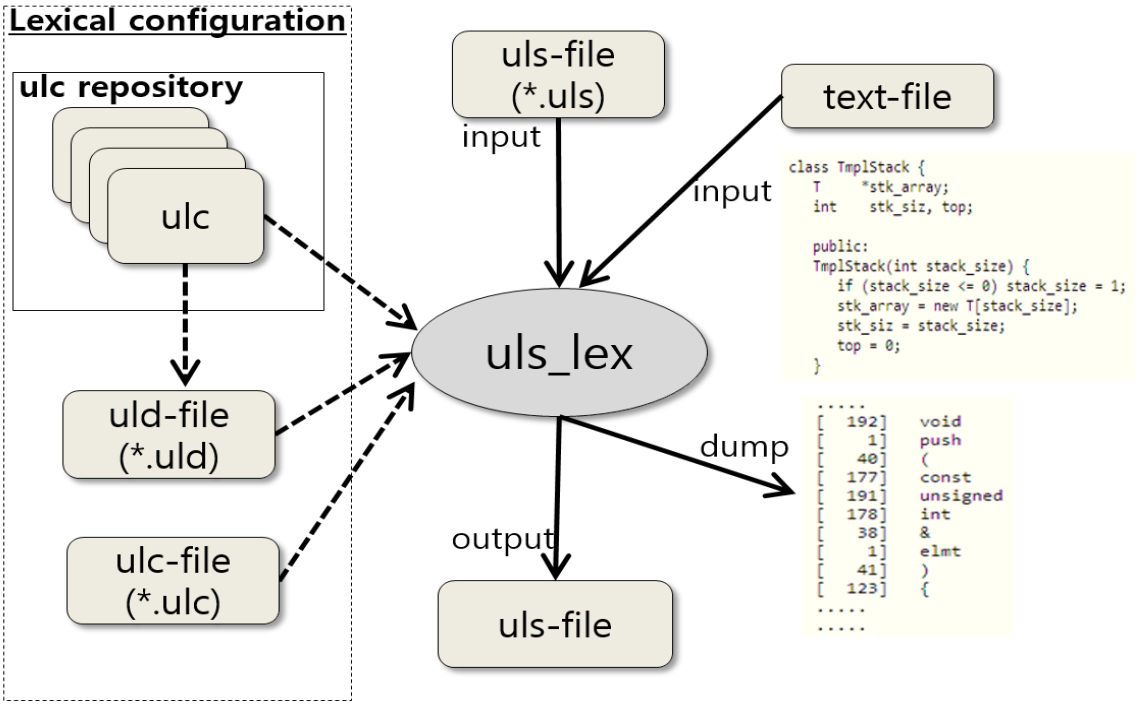
The generated source files are linked with the ULS libraries and your own object files so that the lexical analyzer compliant with your own lexical rules can be embedded into the program. Here is a simple example written in C showing how to use the ULS-library in runtime.

```
#include "sample_lex.h"
//  'sample_lex.h'  must  be  the  header  file  generated  by  ulc2class  from
'sample.ulc'.

uls_lex_t *sample_lex

int main(int argc, char * argv[])
{
    char *input_file = argv[1];
    int tok;

    // create the uls-object from the lexical configuration, 'sample.ulc'.
    sample_lex = uls_create("../adir/sample.ulc");

    // Give 'sample_lex' an input file path to be loaded.
    uls_push_file(sample_lex, input_file, 0);

    for ( ; ; ) {
        tok = uls_get_tok(sample_lex);

        if (tok == TOK_EOI) break;

        // Here, you can use tok, uls_lexeme(sample_lex)
        // as input to syntax analysis.
    }

    // destroy the ulc object 'sample_lex' before exiting.
    uls_destroy(sample_lex);

    // Now the 'sample_lex' is destroyed and won't be referenced.
    return 0;
}
```

4.3. ULC Repository

There's an repository for ulc files so that users can use it to create an object of uls_lex_t type. Also one can edit his ulc-file from scratch and use it by specifying its file path when creating the lexical object.

The tool called 'ulc2class' which generates header files from ulc-file. After including the generated (header) file in your project and building it, you can get a token sequence from input files using the interface of your own lexical analyzer 'uls_lex'. Here a diagram that shows the operations of uls lexical analyzer 'uls_lex'.



To list the available names for ulc in the uls repository, do the followings.

```
$ ulc2class -q
cpp c++ C++
c_sharp C# c# cs csharp c-sharp
go Go golang
visual_basic visual-basic VisualBasic 'visual basic' 'Visual basic'
```

Each line represents the names of a language. You may select any name in same group for your preference. The name should be used as the argument of uls_create() for configuration name. If you want to know whether or not a language is supported by ULS,

```
$ ulc2class -q golang
$ ulc2class -q c++
```

4.3.1. ULC search order

After editing ulc-file, you must position the file in proper directory so that the system can find it. The API uls_create() will search for ulc-file to instantiate a lexical analyzer by reading the file. The search order is different by the format specified in the parameter of uls_create().

It depends on the existence of suffix in the parameter. If you specify it as the name without the suffix, '.ulc', the procedure uls_create() will recognize the argument as a (qualified) ulc-spec-name in the uls

repository. If the parameter constains the suffix(.ulc) or dots('.', '..') or starting directory delimeter, uls_create will recognize it as either relative or absolute file path

The search order for filepath

1. It it starts with '/'. It tries to open it as absolute filepath. If not, it recognized as relative file path.
2. It checks if the environment variable **ULC_PATH** is set, which is delimited by delimeter character.
3. **CWD**: It finds the ulc-file at the current working directory.
4. **EXELOC**: The program file path: the place where the program file is located.
5. It checks the value of ULC_SEARCH_PATH in the sysprops file, the value is the list of directories. The delimeter in the list is ':' and in Linux and ';' in Windows.
6. It checks the directories specified by the constant **ULS_OS_SHARE_DFLDIR** in the source file. It's defined as follow.

   #define ULS_OS_SHARE_DFLDIR "/usr/local/share:/usr/share" in Linux.

   #define ULS_OS_SHARE_DFLDIR "C:₩Program Files₩Common Files₩UlsWin" in Windows.

The search order for ulc-spec-name

1. It checks environment variable **ULS_SPEC_PATH**, which is also delimited by the delimeter character.
2. The ulc system repository: It's related to the installed directory **INST_DIR**.

   INST_DIR/share/ulcs

   The installed directory is specified by the command line 'configure --prefix <INST_DIR>' in linux. In Windows, it's written into the Windows registry value of **ULS_HOME**.

3. ULS_OS_SHARE_DFLDIR: A hard-coded list of directories.

4.4. ULD file

Another lexical configuration file is uld-file suffixed by '*.uld', which is capable of remapping the token name and token number by inheriting the ulc-file in the repository.

After modifying the token numbers and saving them to file suffixed with '.uld', make it generated for tokenizing the source files. The followings illustrate the examples of generating files from uld-file.

```
A simple format of example of using uld-file
$ ulc2class sample.uld
$ ls
  sample.uld SampleLex.cpp SampleLex.h

The -d-option creates the source files in that directory.
$ mkdir src_generated
$ ulc2class -d src_generated sample.uld

The -f-option renames the default filename for the source files.
$ ulc2class -d src_generated -f sample_lex sample.uld
$ ls src_generated
  sample_lex.cpp sample_lex.h

To generates other language of source files,
$ ulc2class -lcs -d src_generated -f samplelex sample.uld
$ ulc2class -ljava -d src_generated -f samplelex sample.uld
$ ulc2class -lc -d src_generated -f samplelex sample.uld
```

To get sample mapping of token name and number, use -s-option with –q-option.

```
$ ulc2class -q -s golang
       #@golang
       ......
       NUMBER -3
       ID -2
       EOF -1
       EOI 0
       RUNE_LITSTR 128
       INTERP_LITSTR 129
       RAW_LITSTR 130
```

You can save and modify token name and number in the above output and save it as uld-file, for example my_go.uld. The file path of 'my_go.uld' should be passed to the argument of uls_create().

5. ULS Keyword Frequency File

The ULS system has an internal keyword search algorithm capable of rearranging the data according to the priority of each keyword. To do this, it's necessary to give the information of frequencies of each keyword. The information is stored in the file with suffix '.ulf'.

The initializer/creator of lexical analyzer will read the ulf-file if the file is located in the same directory as the ulc-file. The system will locate the keyword corresponding to id string, making the cost of searching for keywords to be minimal.

There's a command tool named **ulf_gen** which generates the ulf-file from sample source files. The output of ulf_gen is frequencies for all keywords and is stored in the file suffixed by '*.ulf'. Here's examples of using ulf_gen.

To process all the file paths in listing file 'a.list' in '/package/home'

```
$ ulf_gen -L sample -l a.list /aaa/samples_src
```

To save the output-file to other one than default, use -o-option,

```
$ ulf_gen -o /opt/share/b.ulf -L sample -l a.list /package/home
```

The list file is just a simple text file paths that you want to process. The each line in the file represents the filepath which starts from <target-dir>. For example, the file 'a.list' may be as follows.

```
# Comment here
# A filepath per line
a.c
src/b.c
srclib/c.c
```

6. Token Sequence File

The token sequence from the tokenizer can be temporarily stored in the file called uls-file(*.uls). This file can be again used as an input of the lexical object. The uls-file format is unique one for storing the token sequence, which can be used as intermediate file for generating object file for compilers.

There's a tool named **uls_strem**. It reads source file or lexical stream file(*.uls) and can display it or convert it into another uls-file. In using the tool, It's necessary to specify a lexical configuration from ulc-files(*.ulc) with -L-option. The lexical analyzer will read the input files which can be source files or uls-files. The output file is uls-file, a binary/text token sequence file having a token list with its lexeme. The uls_stream can also dump the token sequence as follow.

```
[     ID]  DDD
[     <=]  <=
[     ID]  EEE
[      {]  YYY
[      =]
```

To dump the token sequence of input1.txt and input1.uls,

```
$ uls_stream -L sample.ulc input1.txt
$ uls_stream -L sample.ulc input1.uls
$ uls_stream -L sample.ulc -o a.txt input1.uls
```

If you specify -C-option with list file, you can get a conglomerate file of all the input files listed in <list-file>. The list file is just a simple list of file paths that you want to process. The each line in the list file stands for the file path which starts from <target-dir>. Let a list file 'a.list' be as follows.

```
# Comment here
# A filepath per line
input1.c
src/input2.c
srclib/input3.c
```

You can run it with -C-option as below.

```
$ uls_stream -C a.list –L sample.ulc /package/home/target-dir
```

The default output-file in binary mode('-b') is the 'a.uls'. The output file is specified with -o-option. When you specify the -L-option, you may consider the search order of ulc-file for the library to properly find it.

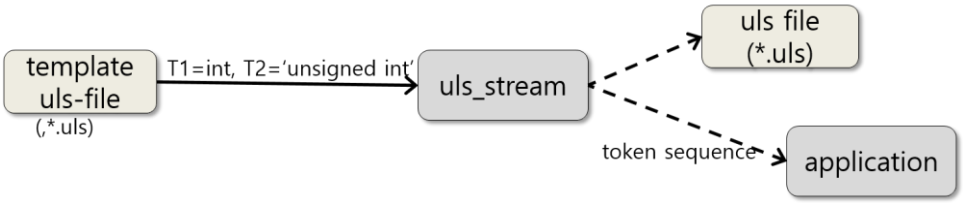
The -f-option can be used with -C-option to filter the files before the input to be passed to lexical analyzer. The argument of -f-option should be a command line which input from stdin and outputs to

stdout like 'gcc -E'.

```
$ uls_stream -b -C a.list -f 'gcc -E' ../package/home
```

6.1.1. Template uls-file

The uls stream file can be used as an intermediate file to provide the token sequence to back-end. The uls-file can have template variables in token sequence. It's instantiated by giving the actual arguments to them. The resultant token sequence can be used as an input of syntax analysis.



To make a template-uls file from a source file tmpl_ex.cc,

```
$ uls_stream -Lsample.ulc -b -TT1 -o tmpl_ex.uls tmpl_ex.txt
   where the value of -T-option 'T1' is the lexeme ID token.
```

To make uls file from the 'tmpl_ex.uls' with the template variables substituted,

```
$ uls_stream -L sample.ulc T1='unsigned int' tmpl_ex.uls
```

Here's a sample input file to test the template input stream.

```
class TmplStack {
       T     *stk_array;
       int    stk_siz, top;

 public:

       TmplStack(int stack_size) {
              if (stack_size <= 0) stack_size = 1;
              stk_array = new T[stack_size];
              stk_siz = stack_size;
              top = 0;
       }

       ~TmplStack() {
              delete[] stk_array;
       }

       void push(const T& elmt) {
              stk_array[top++] = elmt;
       }

       T pop() {
              return stk_array[--top];
       }
```

Let the generated template uls by the 'uls_stream' be 'tmpl_ex.uls'. Then the uls_stream can easily create an uls-file from template uls-file as follows.

```
$ uls_stream -L sample.ulc T='unsigned int' tmpl_ex.uls
.....
[    192]  void
[      1]  push
[     40]  (
[    177]  const
[    191]  unsigned  // the template var 'T has been
[    178]  int       // replaced with 'unsigned int'.
[     38]  &
[      1] elmt
[     41]  )
[    123]  {
.....
.....
```

6.1.2. File Format

The uls-file is a binary file. The file consists of two sections. The size of the first section is fixed as 512-bytes. It has the attributes about stream file.

- NAME: spec name
- CREATION_TIME: The created date/time of this file.
- TAG: The extra name by file creator.

The second section of file is the list of tokens, where each token has the following format.

- token id: a 4-byte integer
- The length of lexeme: a 4-byte integer
- lexeme: A byte string
- '₩0': The end of the lexeme

7. The encodings in ULS

The ULS basically supports for UTF-8 for encoding of all input (text) files on Linux like system. In addition to the default encoding, well-known encoding is supported.

7.1.1. Wide string

The wide string API is supported on both Linux and Windows.

Linux

To enable the wide-string mode on Linux, you must define the constant **ULS_USE_WSTR** when compiling the source files in C/C++ and link your program with the library **libulscompat.so** additionally. Refer the example in the directory 'tests/wdump_toks' for details. The code of ulscompat is self-contained in Uls.dll.

Windows

To enable the wide-string mode on Windows, you only set the '_UNICODE' in 'the project property popup windows'. The macro constant **ULS_USE_WSTR** in ULS is set according to the constant '_UNICODE' in Visual Studio.

All the procedures in ULS related to wide string is redirected to the wide string specific API in 'ulscompat' including uls_lexeme(), uls_get_tag(), and the like.

In short, to create program using wide-string uls API,

A. Define the macro constant **_UNICODE** when compiling it. The constant **ULS_USE_WSTR** is automatically defined by checking the charset, **_UNICODE**, in the project properties in Visual Studio.
B. Link your program additionally with **libcompat.so** if you're in Linux environment.

7.1.2. MS-MBCS Encoding

The 'MS-MBCS Encoding' is the multibytes encoding in Windows. The MS-MBCS Encoding in ULS is not any more supported. The default Encoding mode is utf8. To use this , set the option of the visual studio configuration of your project.

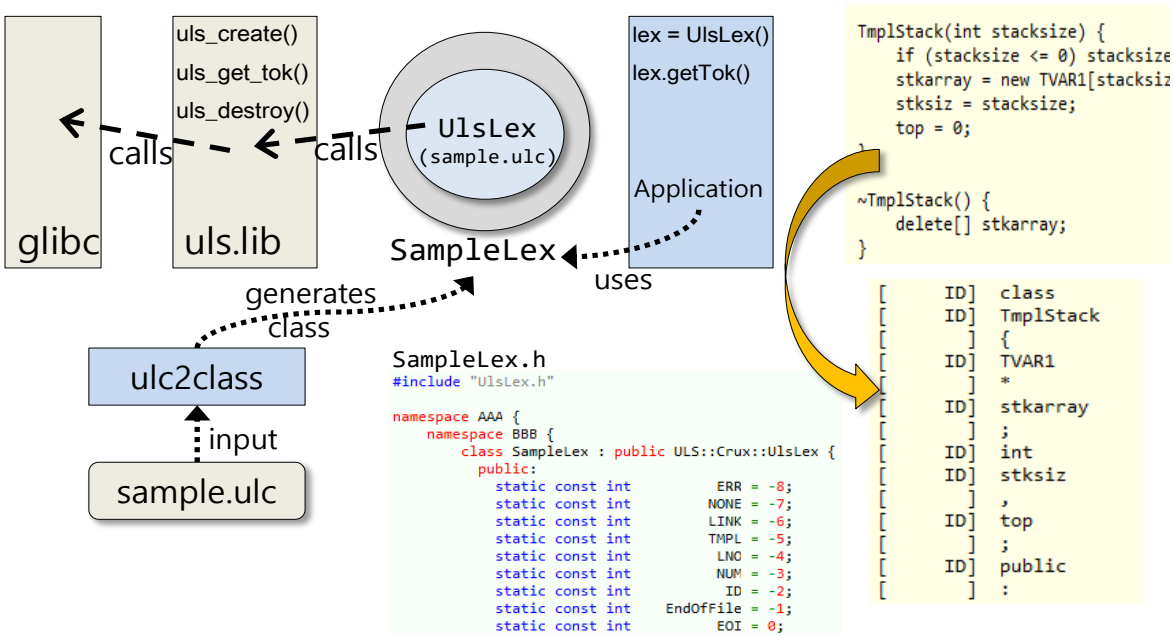
7.2. The Design of ULS Class library

The class library is consists of the classes that can interface with the request of users. There are 3 main

classes implemented in the ULS class library.

- ✧ UlsLex
- ✧ UlsIStream
- ✧ UlsOStream

There exist classes with same name and similar function in the other supported languages, C#, Java. The classes have been implemented to wrap the functions in the C or C++.NET libraries. Here is a diagram to illustrate the process of tokenizing the source files.



To begin with, a proper ulc file must be selected. Above sample.ulc is processed by the program 'ulc2class' to generate the header file 'SampleLex.h'. The file has token definitions for each token and inherits the class 'UlsLex' in the c++ library.

Then user includes the class definition file 'SampleLex.h' in his sources. Using the API of UlsLex via SampleLex, one is able to tokenize input files. The resultant token sequence should be mostly an input of syntax analysis.

8. Examples

The source code in 'tests' and 'examples' can be used to see the scenarios using ULS API.

8.1. Tests

There're example directories in 'tests' where programs are executed by issuing command *'make check'*. The source files of each directory are the examples using API in order to test ULS features.

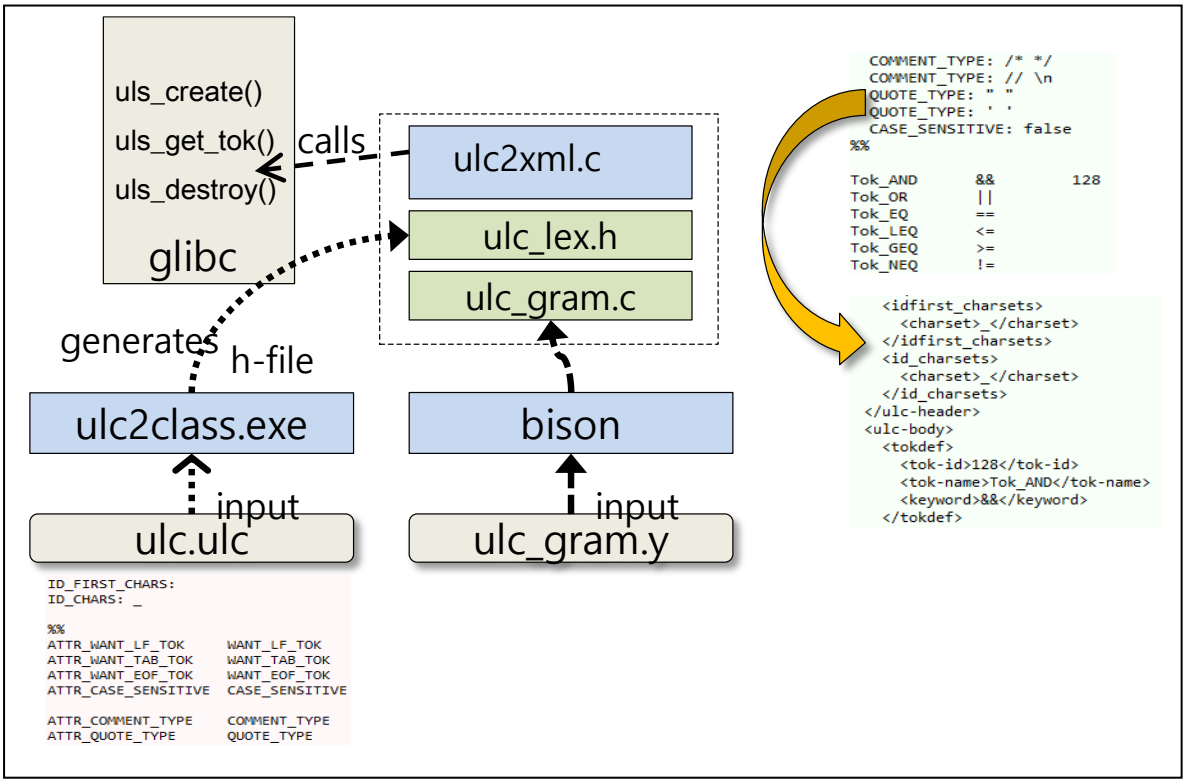
- **dump_toks**: A basic use of ULS just dumping the tokens in input files.
- **getch**: The ULS supports getting/ungetting one char in input-file, using uls_getch(), uls_ungetch(), or uls_peekch()
- **id_stat**: This shows the statistics of identifiers from input-files, recognizing the IDs.
- **line**: You can give a literal line as input of the uls-object by calling uls_set_line() or uls_push_line() API.
- **log**: The ULS has a unique logging system(uls_lf_) with vararg-style. With the logging system you can log error-messages including the location of error(%w), token string(%k) and token name(%t). The 'log' demonstrates how to use the uls_lf.
- **many_kwrds**: The ULS supports the languages having many keywords with little loss of performance.
- **nest**: It demonstrates the languages that contain a sub-language in it by instantiating multiple of ULS-objects.
- **strings**: The ULS can extract the literal strings from input files.
- **tok_remap**: The ULS can remap the token numbers of 1-char ACSII, such as '(', '{', '=', ... to arbitrary integers.
- **two_lex**: An example using two ULS-objects.
- **cpp_hello**: It demonstrates the usage of ULS-class in C++, using the libulscpp.*. This is also contained in the Windows version as an example.

8.2. Demos

There're also demo programs in the 'examples' directory which are linked with C++ library. The programs in 'examples' are built with the installed ULS libraries.

8.2.1. yacc

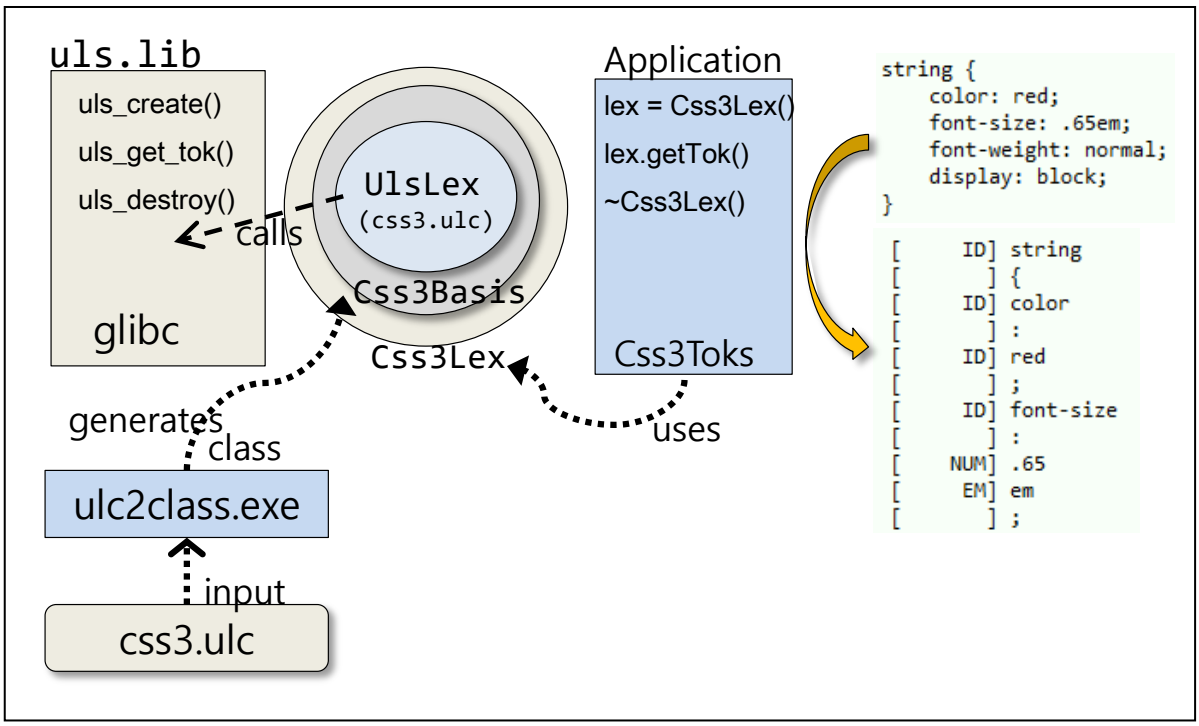
The ULS can be used as a replacement of 'lex/flex' by providing token sequence to yacc/bison parser. The program 'ulc2xml' uses the program bison to link the output of uls with it. An ulc style file is input file and the output is the converted xml file. This is an illustration to show that the lexical analyzer produced by ULS can be used with yacc/bison linked.



'ulc.ulc' a rough and old description of ulc file and 'ulc_gram.y' is an yacc grammar file. The executable is a program linked with the 'uls' library. The file 'ulc.ulc' is processed by the program 'ulc2class' to generate the header file 'ulc_lex.h'. The file has macro definitions for the token definitions.

8.2.2. Css3

'Css3' scans css3 file and dump its tokens to the standard output.

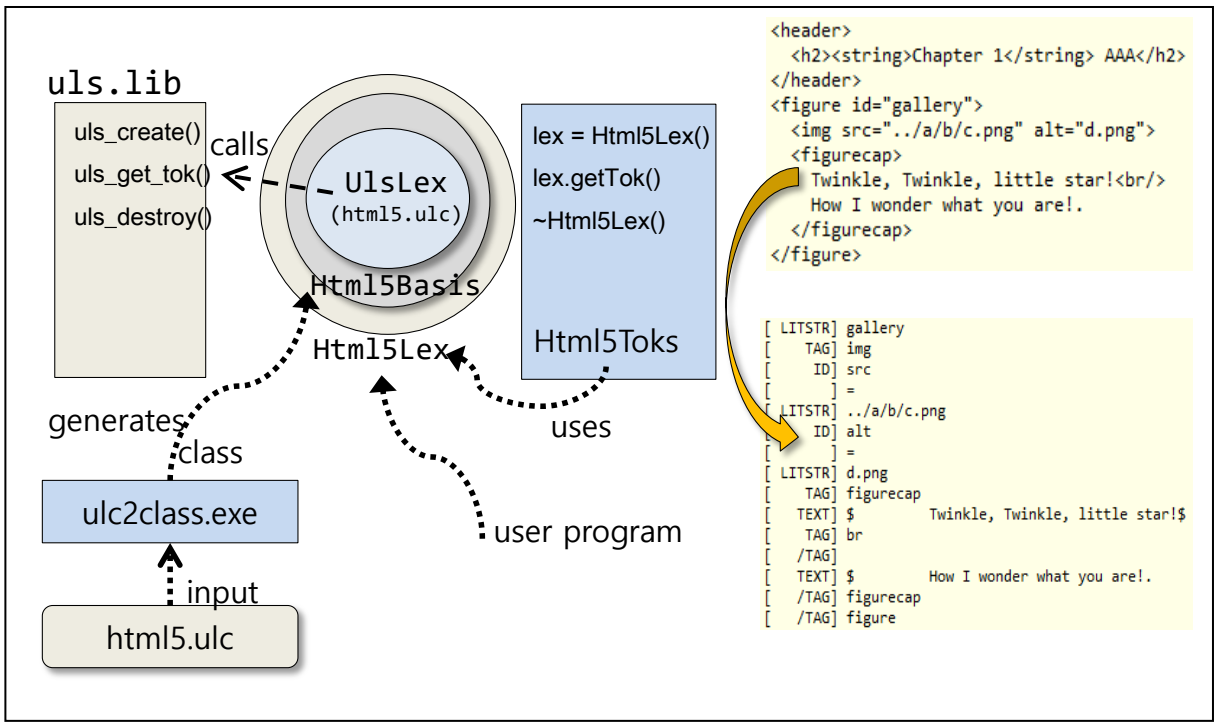


'css3.ulc' is selected to tokenize css3 file and the ulc file is processed by the program 'ulc2class' to generate the header file 'Css3Basis.h'. The file has token definitions for each css3 token and inherits the class 'UlsLex' in the c++ library.

The class Css3Lex inherits Css3Basis. Using the API of UlsLex via Css3Basis, it tokenizes the input file. The input is a css3 file and the output is a token sequence.

8.2.3. Html5

This demo program scans html file and dump its tokens to the standard output



Above html5.ulc is processed by the program 'ulc2class' to generate the header file 'Html5Basis.h'. The file has token definitions for each token and inherits the class 'UlsLex' in the c++ library.

The class Html5Lex inherits Html5Basis. Using the API of UlsLex via Html5Basis, it will tokenize the input file. The input is a html file and the output is a token sequence. The resultant token sequence will be usually an input of syntax analysis.

8.2.4. Shell

Scans shell-script and dump its tokens to the standard output.

8.2.5. Mkf

Scans 'Makefile' file and dump its tokens to the standard output

8.2.6. tokseq

Tests the token stream API, writing and reading uls-files

9. Using the libraries

This chapter includes the detailed descriptions regarding each ULS function and its associated data types in the library 'libuls'(or Uls.dll). This chapter explains the programming API in C. Below are defined the alternative typenames for the frequently used data types.

- **ConstString:** It represents a read-only string and it is pointer of type 'const char'.
- **String :** pointer of type 'char'. The content of it can be written.
- **Boolean** : 'int'. The value is 1 for true and 0 for false.
- **SystemError**: represents 'int' type that the value has 0 for success, negative-integer for failure.
- **OpaqueData** : pointer of void
- **BitFlags :** the type 'int' considered as bit fields.

For different languages, the above type names are defined as follows.

	C	C++	C#	Java
ConstString	const char *	string	string	string
String	char *	string	string	string
Boolean	int	bool	bool	bool
SystemError	int	int	int	int
OpaqueData	void *	void *	object	object

In the followings, 'uls' is an object, pointer of 'uls_lex_t' type for lexical analysis, which is created by uls_create() or uls_init().

9.1. The Lexical Object

This section explains the programming API on the main structure 'uls_lex_t'.

♦ `uls_init(uls, confname)`

This initializes the empty structure 'uls' with the lexical configuration read from 'confname'. The 'confname' may be a name of language specification in the uls repository or simply a file path of 'ulc' file. To see the available ulc names, use the command ulc2class with –q-option. Don't forget to de-initialize the object 'uls' using the uls_deinit() after working with it.

confname : A file that has a lexical configuration with ulc-file format (ConstString)

RETURN : (SystemError)

♦ `uls_deinit(uls)`

The counterpart of uls_init(), which de-initializes 'uls', releasing the memories used by 'uls'.

♦ `uls_create(confname)`

To tokenize input file, It's necessary create a lexical analyzer object. To do this, you can use uls_init() or uls_create(). The uls_init() can be used to initialize a static data type. The 'uls_create()' does for itself allocate the memory of type 'uls_lex_t' and initialize by calling 'uls_init().

confname : A file that has a lexical configuration with ulc-file format. (ConstString)

RETURN : **(uls_lex_t*)** the pointer of structure type uls_lex_t

♦ `uls_destroy(uls)`

The counterpart of uls_create(), this de-initializes the 'uls' and deallocates the memory of 'uls'.

♦ `uls_tok(uls)`

The token number of current token as defined in the corresponding ulc-file. The program uses the values in the (token) number in header file, enum-structure, or class constants. When the input is exhausted, the last token is EOI, End Of Input.

RETURN : **(int)** The current toke number

SYNONYM : uls_tokid, uls_toknum, uls_token

♦ `uls_lexeme(uls)`

The current token string is returned. The string as 'const char *' is valid only before calling for the next token. So, the value of this should be copied if necessary. The string encoding is utf-8. But if you enable the other encodings(MS-MBCS, wide-string), you can use the value of uls_lexeme() as the string of the encoding.

RETURN : **(ConstString)** The token string of current token.

SYNONYM : uls_tokstr

♦ `uls_lexeme_len(uls)`

The length of the current token string, uls_lexeme(uls). It's the number of bytes of the utf8 string. If the MS-MBCS mode is set on, it's the number of bytes of the MBCS string..

RETURN : (int) the (byte) length of the current token string

♦ `uls_lexeme_chars(uls)`

Use this to get the number of characters of the current token string, uls_lexeme(uls). The size of character depends upon the used encoding. If the MS-MBCS mode is on. The size is variable by 1 or 2 of bytes. If the wide-string encoding mode is on, the size is sizeof(wchar_t), maybe 4.

RETURN : (int) the length of unicode chars

♦ `uls_is_eoi(uls)`

It checks whether the cursor of input is at the end of input or not. If it's true, the current token should be EOI, End of Input.

RETURN : (Boolean)

♦ `uls_is_eof(uls)`

It returns true if the input cursor is pointing the end of file. As the input for lexical analysis may have multiple nested input files, you can get multiple EOF-tokens at the end of each input source. You get an EOF-token only if you set the flag ULS_WANT_EOFTOK in the call of uls_push_*(). See the example the procedures in tests/dump_toks for details.

RETURN : (Boolean)

♦ `uls_is_err(uls)`

It returns true if the uls object 'uls' is in state of error like undefined token or incorrect input I/O.

RETURN : (Boolean)

♦ `uls_is_id(uls)`

It returns true if the current token is an identifier.

RETURN : (Boolean)

♦ `uls_is_number(uls)`

It returns true if the current token is a integer or floating number

RETURN : (Boolean)

♦ `uls_is_linenum(uls)`

It returns true if the current token has line information.

RETURN : (Boolean)

♦ `uls_is_tmpl(uls)`

It returns true if the current token is the token of type template variable. The lexeme of it is the name of the template variable.

RETURN : (Boolean)

♦ `uls_toknum_eoi(uls)`

The token number of the object 'uls' is varied for each configuration. This function returns the number of the reserved token EOI associated with the 'uls'.

RETURN : (int)

♦ `uls_toknum_eof(uls)`

The token number of the object 'uls' is varied for each configuration. The function returns the number of the reserved token EOF associated with the 'uls'.

RETURN : (int)

♦ `uls_toknum_err(uls)`

The token number of the object 'uls' is varied for each configuration. The function returns the number of the reserved token ERR, which is associated with the 'uls'.

RETURN : (int)

♦ `uls_toknum_id(uls)`

The token number of the object 'uls' is varied each configuration. The function returns the number of the reserved token ID associated with the 'uls'.

RETURN : (int)

♦ `uls_toknum_number(uls)`

The token number of the object 'uls' is varied for each configuration. The function returns the number of the associated reserved token NUMBER.

RETURN : **(int)**

♦ `uls_toknum_linenum(uls)`

The token number of the object 'uls' is varied for each configuration. The function returns the number of the associated reserved token LINENUM.

RETURN : **(int)**

♦ `uls_toknum_tmpl(uls)`

The token number of the object 'uls' is varied for each configuration. The function returns the number of the associated reserved token TMPL.

RETURN : **(int)**

♦ `uls_get_lineno(uls)`

This returns the current line number of the input stream. The line number of 'uls' represent the location in input that is currently processed.

RETURN : **(int)** the line number of input source file.

♦ `uls_get_tag(uls)`

This returns the stored tag in 'uls'.

ptr_len : (int *) OUT-PARAM — Use uls_get_taglen(uls) to retrieve the length of the returned string.

RETURN : **(ConstString)** The current tag of the input stream. The tag can be changed by uls_set_tag().

SYNONYM :

♦ `uls_set_tag(uls, tagstr, lno)`

It can set the tag of the current input stream. More often than not, the 'tagstr' is the filepath in case the current input stream is a file.

tagstr : (ConstString) — 'tagstr' is the string that you can retrieve it from uls_get_tag(). It's the tag that the current input represents itself. The pair <tagstr,lno> constitutes the current coordinate of the current input.

lno : (int) The parameter 'lno' is the line number and the location in the input. It accept non-negative integer. If a negative value is passed, it's ignored.

♦ `uls_set_lineno(uls, lno)`

This is a abridged version of uls_set_tag() to set only the line number of the current input. Usually You don't have to call this procedure because the system of 'uls' automatically adjusts the line number for itself.

lno : (int) 'lno' is the line number and the location in the input. It accepts non-negative integer. If negative value is passed, it's ignored.

♦ `uls_inc_lineno(uls, n_delta)`

It adds 'n_delta' to the line number of the current input in 'uls' by 'n_delta'. It may be negative. If the resultant value is negative, the procedure to update it is cancelled.

n_delta : (int) an integer which is added to the current line number in 'uls'

♦ `uls_tok2keyw(uls, tok_id)`

The keyword string corresponding to 'tok_id'. For example, if the tok_id is 1 and it represents the keyword '+=', this procedure returns the string '+='.

tok_id : (int) The token id of which keyword string you want to know.

RETURN : (ConstString)

♦ `uls_tok2name(uls, tok_id)`

This returns the name of 'tok_id'. For example, if the tok_id is 1 and it represent the keyword '{', this procedure might return the string 'BEGIN'.

tok_id : (int) The token id of which name you want to know.

RETURN : (ConstString)

♦ `uls_push_file(uls, filepath, flags)`

This method puts the input-file of the path 'filepath' onto the internal input stack.

filepath : (ConstString) The 'filepath' can be utf8, MS-MBCS, or wide-string according to the used encoding. The basic encoding of ULS is utf8 in Linux and MS-MBCS or 'MS Code Pages' in Windows.

flags : **{ ULS_WANT_EOFTOK | ULS_DO_DUP }**
The flag **ULS_WANT_EOFTOK** allows the user to get the token EOF at the end of FILE. By calling uls_pop(), you enable the remaining content of current input to be dismissed and the next input under the current one of the input stack to be prepared. The flag **ULS_DO_DUP** is to copy the file of 'filepath' into another place. The user is free to read, write, or delete the file specified in the parameter.

RETURN : **(SystemError)** Usually if the 'filepath is not found or the user hasn't sufficient permission to use it, it returns a negative integer.

♦ `uls_push_line(uls, line, len, flags)`

This method pushes 'char' string type input onto the input stack.

line : (ConstString) The file path of input.

flags : **{ ULS_WANT_EOFTOK | ULS_DO_DUP }**
The flag **ULS_WANT_EOFTOK** allows the user to get the token EOF at the end of FILE. It's possible to call uls_pop() so that the next input under the current one of the input stack can be prepared. The flag **ULS_DO_DUP** is to copy the string line so that the user can release the original memory of 'line'.

len : (int) If you pass -1 to it, it will get the length of 'line', recognizing 'line' as a null terminated string.

RETURN : **(SystemError)**

♦ `uls_push_fd(uls, fd, flags)`

This pushes the input source 'fd', a file descriptor, onto the current input of uls. The current input is not dismissed but postponed. After the end of processing the input-stream, the previous input-stream will be returned.

fd : (int) The input stream that provided by user. By default, the file descriptor 'fd' is NOT duplicated(by linux syscall dup()) for safe use.

flags : **{ ULS_WANT_EOFTOK | ULS_DO_DUP }**
Set ULS_WANT_EOFTOK if you want EOF-token at the end of input specified by 'fd'. By default, the file descriptor 'fd' should not be touched until the IO for tokenizing is finished. But if you set the flag **ULS_DO_DUP,** then the file descriptor 'fd' will be duplicated by system call. In this case, the original 'fd' can be freely closed.

RETURN : **(SystemError)**

♦ `uls_push_fp(uls, fp, flags)`

This pushes the input-stream, file pointer, onto the current input-stream stack of uls. The current input-stream is not dismissed but postponed.

fp : The file pointer 'fp' that may have been opened by fopen() in stdio.h (FilePointer)

flags : **{ ULS_WANT_EOFTOK | ULS_DO_DUP }**
Set ULS_WANT_EOFTOK if you want the EOF-token at the end of input of 'fp. If you set the flag **ULS_DO_DUP,** the file content indicated by 'fp' is copied to the temporary directory and used for 'uls'.

RETURN : **(SystemError)**

♦ `uls_pop(uls)`

This is used to dismiss the current input in the top of the input stack in 'uls'. The input pointer of uls is returned to the previous position of the input.

SYNONYM : uls_dismiss_input

♦ `uls_pop_all(uls)`

This procedure dismisses all the input-streams on the internal input stack of uls.

SYNONYM : uls_dismiss_inputs

♦ `uls_peek_uch(uls, tok_peek)`

Normally, this procedure returns the next char of the input. In case of the input pointer being at the front of a literal string, EOF, EOI, NONE, it returns a special constant **ULS_UCH_NONE** and the token number of the literal string into the integer buff pointed by 'tok_peek'. The uls_peek_ch() just peek the next character without advancing the input cursor.

tok_peek : The pointer of uls_nextch_detail_t can be given to get token information. OUT-PARM

RETURN : **(uls_uch_t)** The unicode which is at the position of the input cursor.

SYNONYM : uls_peekch

♦ `uls_get_uch(uls, tok_peek)`

The current character to which the input cursor is pointing will be returned. The procedure will advance the cursor to next position. If the cursor is at a literal string, EOF, EOI, or NONE, it returns **ULS_UCH_NONE**.

The uls_get_uch can't break literal string to characters. When this procedure comes accros a literal string, the information of the string is returned in the parameter 'uch_detail'.

- ✓ uch_detail.tok_id : the token number if the current token is a quotation string, EOF, EOI, or NONE otherwise.
- ✓ uch_detail.is_quote: true if the current position indicated the literal string false otherwise.

In case the current token is a literal string, you can use the push-line interface to break the string into characters.

tok_peek : The pointer of uls_nextch_detail_t can be given to get token information.
OUT-PARM

RETURN : **(uls_uch_t)** The unicode which is at the position of the input cursor.

SYNONYM : uls_getch

♦ `uls_get_tok(uls)`

This is one of the main procedure of uls lexical analyzer, which is created by uls_create() or uls_init(). It gets a token and its associated lexeme advancing the cursor of input. The reserved EOI token number will be returned at the end of input. Even if it returns the token number, you can get the same value by the procedure uls_tok(). The associated token string is obtained by uls_lexeme().

RETURN : **(int)**

SYNONYM : uls_gettok

♦ `uls_set_tok(uls, tokid, lexeme, l_lexeme)`

It replaces forcibly the current token with the user provided token.

tokid : The token id assigned to 'lxm'
(int)

lexeme : The string to be the current token string
(ConstString)

l_lexme : The length of 'lexeme'
(int)

SYNONYM : uls_settok

♦ `uls_dump_tok(uls, pfx, suff)`

This dumps the current token information in one line. The line is composed of the string "pfx", "suff", and the token. For example, if pfx is '₩t' and suff is '₩n', the output line to stdout may

be like as follows.

'₩t' **[___ID] main** '₩n'

pfx : The prefix string of the line.
(ConstString)

suff : The suffix string of the line.
(ConstString)

♦ `uls_expect(uls, ExpectedTok)`

This checks whether the current token is same as the parameter 'expected' or not. If it's not, the program will be aborted.

ExpectedTok : The token id to be expected.
(int)

♦ `uls_set_extra_tokdef(uls, tok_id, extra_tokdef)`

This method appends the extra information to the token definition, name and keyword, associated with 'tok_id'. The extra information can be anything, such as precedence, node-id.

tokid : The token id of which extra token definition will set to 'extra_tokdef'.
(int)

extra_tokdef : An anonymous data given by user for the token.
(OpaqueData)

♦ `uls_get_extra_tokdef(uls, tok_id)`

Returns the user data corresponding to the token id 'tok_id'.

tokid : The token id for which you want its extra token definition.
(int)

RETURN : **(OpaqueData)**
The user provided data previously hooked up by uls_set_extra_tokdef().

SYNONYM : uls_extra_tokdef

♦ `uls_set_current_extra_tokdef(uls, extra_tokdef)`

This is an abridged version of uls_set_extra_tokdef(). In short this calls uls_set_extra_tokdef() with the current token number.

extra_tokdef : An opaque data given by user for the token.
(OpaqueData)

♦ `uls_get_current_extra_tokdef(uls)`

This returns the user data connected to the current token (number). The user data is the one linked by calling uls_set_extra_tokdef().

RETURN : **(OpaqueData)**
The user provided data previously hooked up by 'uls_set_extra_tokdef'

SYNONYM : uls_current_extra_tokdef

♦ `uls_unget_str(uls, str)`

This will retreat the cursor of input stream by the length of 'str' pushing the string 'str'.

str : The string to be pushed
(ConstString)

♦ `uls_unget_lexeme(uls, lxm, tok_id)`

This will push an white character and retreat the cursor as the length of 'str'

lxm : The string to be pushed
(ConstString)
tokid : The token id assigned to 'lxm'
(int)
SYNONYM : uls_ungetch

♦ `uls_unget_tok(uls)`

This will cancel the current token and the next call of uls_get_tok will return it again.

SYNONYM : uls_ungettok

♦ `uls_unget_ch(uls, ch)`

This will retreat the cursor of the input stream to the one previous position and put 'ch' in it.

ch : The char to be pushed.
(char)
SYNONYM : uls_ungetch

♦ `uls_is_int(uls)`

It checks whether the current token is of type integer or not. Notice the format of the lexeme for integer token is string of hexadecimal characters without any prefix, such as '0x' or 'h'.

RETURN : (Boolean)

SYNONYM :

♦ `uls_is_real(uls)`

This checks the current token is of type floating number. The format of the lexeme for floating number is one of the followings.

a) .[1-9][0-9]*

b) .[1-9][0-9]*E[-][1-9]+

c) .0

RETURN : (Boolean)

SYNONYM :

♦ `uls_is_zero(uls)`

It checks whether the current token is zero or not regardless of whether it's int or floating.

RETURN : (Boolean)

♦ `uls_lexeme_int32(uls)`

It returns a 32-bits integer from the current token string. If the current token is not number token it'll return an incorrect value. Make sure the current token is a number token.

RETURN : (uls_int32) The binary format of 'int' of the lexeme of current token.

♦ `uls_lexeme_uint32(uls)`

It returns a 32-bits unsigned integer from the current token string. If the current token is not number token of excess the capacity of unsigned integer, it'll return an incorrect value. Be sure the current token is a number token.

RETURN : (uls_int32) The binary format of 'int' of the lexeme of current token.

♦ `uls_lexeme_int64(uls)`

It returns a 64-bits integer from the current token string. If the current token is not number token it'll return an incorrect value. Make sure the current token is a number token.

RETURN : (uls_int64) The binary format of 'int' of the lexeme of current token.

♦ `uls_lexeme_uint64(uls)`

This returns a 64-bits unsigned integer from the current token string. If the current token is not number token it'll return an incorrect value. Make sure the current token is a number token.

RETURN : **(uls_uint64)** The binary format of 'int' of the lexeme of current token.

♦ `uls_lexeme_int(uls)`

If the current token is number and the interpreted lexeme is in the range of 'int' it'll return the correct value of the lexeme in the format of 'int'.

RETURN : **(int)** The binary format of 'int' of the lexeme of current token.

SYNONYM : uls_lexeme_d

♦ `uls_lexeme_uint(uls)`

If the current token is number and interpreted lexeme is in the range of 'unsigned int' it'll return the correct value of the lexeme in the format of 'unsigned int'.

RETURN : **(unsigned int)** The binary format of 'unsigned int' of the lexeme of current token.

SYNONYM : uls_lexeme_u

♦ `uls_lexeme_long(uls)`

If the current token is number and interpreted lexeme is in the range of 'long' it'll return the correct value of the lexeme in the format of 'long'.

RETURN : **(long)** The binary format of 'long' of the lexeme of current token.

SYNONYM : uls_lexeme_ld

♦ `uls_lexeme_ulong(uls)`

If the current token is number and interpreted lexeme is in the range of 'unsigned long' it'll return the correct value of the lexeme in the format of 'unsigned long'.

RETURN : **(unsigned long)** The binary format of 'unsigned long' of the lexeme of current token.

SYNONYM : uls_lexeme_lu

♦ `uls_lexeme_longlong(uls)`

If the current token is number and interpreted lexeme is in the range of 'long long' it'll return the correct value of the lexeme in the format of 'long long'.

RETURN : **(long long)** The binary format of 'long' of the lexeme of current token.

SYNONYM : uls_lexeme_lld

♦ `uls_lexeme_ulonglong(uls)`

If the current token is number and interpreted lexeme is in the range of 'unsigned long long', it'll return the correct value of the lexeme in the format of 'unsigned long long'.

RETURN : **(unsigned long long)** The binary format of 'unsigned long long' of the lexeme of current token

SYNONYM : uls_lexeme_llu

♦ `uls_lexeme_double(uls)`

If the current token is number and interpreted lexeme is in the range of 'double' it'll return the correct value of the lexeme in the format of 'double'.

RETURN : **(double)** The binary format of 'double' of the lexeme of current token.

SYNONYM :

.

9.2. ULS Lexical Stream

9.2.1. Input Stream

Uls input stream is an object to manipulate sequential read only files.

- `uls_init_tmpls(tmpl_list, n_alloc, flags)`

  This procedure is to initialize the structure 'tmpl_list' of type uls_tmpl_list_t. There's struct uls_tmpl_t that implements a template variable, which consists of name and value. The structure 'uls_tmpl_list_t' is the list of the pairs.

  tmpl_list : (uls_tmpl_list_t*) OUT-PARAM — The pointer of target structure that should be initialized. The structure consists of the list of template variables.

  n_alloc : (int) — The capacity of template variables.

  flags : (BitFlags) — The memory for the name and its value is by default referenced. To duplicated the memory use the flag **ULS_TMPLS_DUP.**

  RETURN : **(SystemError)** Returns a negative value if the procedure fails

- `uls_deinit_tmpls(tmpl_list)`

  It lets the allocated memory for 'tmpl_list' be released. But the memory for 'tmpl_list' itself is not released.

  tmpl_list : (uls_tmpl_list_t*) .The target object to be memory-deallocated.

  RETURN : **(SystemError)**

- `uls_create_tmpls(n_alloc, flags)`

  This allocates the memory for a structure 'uls_tmpl_list_t' and initializes it by calling uls_init_tmpls(). To release the returned memory, use uls_destroy_tmpls().

  n_alloc : (int) — The capacity of template variables.

  flags : (BitFlags) — The memory for the name and its value is by default referenced. To duplicate the memory use the flag **ULS_TMPLS_DUP.**

  RETURN : **(uls_tmpl_list_t*)**

- `uls_destroy_tmpls(tmpl_list)`

  This deallocates the memory allocated by `uls_create_tmpls()`.

tmpl_list : . The template list object to be memory-deallocated.
(uls_tmpl_list_t*)

RETURN : **(SystemError)**

♦ `uls_get_tmpl_value(tmpl_list, name)`

This returns the value of template variable named 'name' in 'tmpl_list'. If there is no template variable named 'name', it returns NULL pointer.

tmpl_list : A list of template variables to be searched.
(uls_tmpl_list_t*)

name : The name of template variable of which we want to know the value.
(ConstString)

RETURN : **(ConstString)** the (string) value of the template variable

♦ `uls_set_tmpl_value(tmpl_list, name, val)`

This procedure is used to modify the value of template variable if it exists. Notice that a template variable is composed of a name and its (string) value.

tmpl_list : The target list of template variables to be modified.
(uls_tmpl_list_t*)

name : The name of the template variable.
(ConstString)

val : New string value of 'name'.
(ConstString)

RETURN : **(SystemError)**

♦ `uls_add_tmpl(tmpl_list, name, val)`

To add a pair of name and its value, use this procedure. Notice that a template variable is composed of a name and its (string) value.

tmpl_list : . The target list of template variables to be modified.
(uls_tmpl_list_t*)

name : The name of the template variable.
(ConstString)

val : The string value of 'name'.
(ConstString)

RETURN : **(SystemError)**

♦ `uls_open_istream(fd)`

This opens an input stream from a file descriptor 'fd'. The returned pointer of input stream can be pushed onto the top of input stack of uls object as follows.

uls_push_istream(istr, flags);

It is closed the input stream in a safe manner only if it's pushed.

uls_close_istream(istr);

The procedures where the opened 'istr' is used are the followings.

- ✧ uls_set_istream_tag
- ✧ uls_push_istream
- ✧ uls_start_stream

fd : A file descriptor to be used as an input
(int)

RETURN : **(uls_istream_t*)** A pointer of input stream structure

♦ `uls_open_istream_file(filepath)`

It opens an input stream from the path of 'filepath' and returns a pointer of input stream.

filepath : The file path to be opened.
(ConstString)

RETURN : The file header having information on 'filepath'
(uls_istream_t*)

♦ `uls_open_istream_fp(fp)`

This creates and returns an input stream structure initialized from a file pointer.
The file pointer might be opened by fopen().

fp : A file pointer opened by fopen()
(FilePointer)

RETURN : **(uls_istream_t*)** Returns NULL if it fails.

♦ `uls_close_istream(istr)`

It flushes the buffer in 'istr' and closes it. Afterwards the pointer 'istr' shouldn't be used.

istr : The opened input stream.
(uls_istream_t*)

RETURN : **(SystemError)**

♦ `uls_set_istream_tag(istr, tag)`

This is used to attach the tag to the opened input stream 'istr'. The 'tag' is a string, which is to be stored in the uls file of 'istr' as the attribute 'TAG:'.

istr : The opened input stream.
(uls_istream_t*)

tag : The string has a limited size 31.
(ConstString)

♦ `uls_push_istream(uls, istr, tmpls, n_tmpls, flags)`

The input-source in ULS can be filepath, file pointer, string, file descriptor, uls_istream_t. This function pushes a input source of type uls_istream_t onto the input-stream stack of uls..

uls : The lexical object
(uls_lex_t*)

istr Input stream
(uls_istream_t*)

tmpls : An uls_tmpl_t represents a template variable. So a multiple of template variables
(uls_tmpl_t*) are passed to this procedure so that the template variables in 'istr' can be replaced with actual values.

n_tmpls : The size of array 'tmpls'.
(int)

flags : The 'flags' is a same meaning as uls_push_file().

RETURN : (SystemError)

9.2.2. Output Stream

Uls output stream is an object to manipulate sequential write only files.

♦ `uls_create_ostream(fd, uls, subname)`

It creates an object supporting write-only sequential IO to save the token sequence generated from the object 'uls'. It contains some information to manipulate the file(stream) 'fd'. The returned value of this procedure can be used until the uls_close_istream() is called.

fd : The file descriptor for output file.
(int)

uls : The lexical object
(uls_lex_t*)

subname The name to be stored in the output stream file. This is the value of the
(ConstString) attribute 'TAG:' in the output file(*.uls).

RETURN : (uls_ostream_t*) It's the uls-header for output file. It returns NULL if it fails.

♦ `uls_close_ostream(ostr)`

This closes the output stream that is opened for streaming I/O.

ostr : The open output stream to be closed
(uls_ostream_t*)

RETURN : **(SystemError)**

♦ `uls_print_tok(ostr, tokid, tokstr)`

It prints a record of <tokid, tokstr> pair to the output stream 'ostr'.

tokid : token id to be printed
(int)

tokstr : the token string associated with the 'tokid'
(ConstString)

RETURN : **(SystemError)**

♦ `uls_print_tok_linenum(ostr, lno, tag)`

It prints an annotation <linenum, tag> pair. When the output file is reread these tokens contribute to automatically update the data of the lexical object, the tag and linenum. The tag can be retrieved by uls_get_tag() and the linenum by uls_get_lineno().

lno : the line number of the source file
(int)

tag : the tag of the source file.
(ConstString)

RETURN : **(SystemError)**

♦ `uls_start_stream(ostr, flags)`

This starts streaming the token sequence in 'istr' into 'ostr'

ostr The output stream pointer to be written
(uls_ostream_t*)

flags : The flag **ULS_LINE_NUMBERING** can be used to insert the line number token whenever the line number is changed in the source.

RETURN : **(SystemError)**

9.3. Logging Framework

Logging Framework supplies with an unique (varargs-style) logging framework that can be used for general purposes. It is newly written from scratch. The common conversion specification %s, %c, %d, %u, %f, etc is not only supported a few of new ones also supported. Thus, It enables the users to have their own formatted *-printf family by adding or overriding conversion specifications. The conversion specification that can be used in format string is as follows.

- **%s**: The corresponding argument must be a string, pointer of char.

  The corresponding string of '%s' must be utf8-encoded string in Linux and MS-MBCS string in Windows.
- **%c**: an ASCII character
- **%d**: an integer of type 'int'
- **%u**: an unsigned integer of type 'unsigned int'
- **%f, %e**: a floating number of type 'double' or 'float'.
- **%lf, %le**: The 'long double' type.
- **%ld**: an integer of type 'long'
- **%lu**: an integer of type 'unsigned long'
- **%lld**, **%Ld**: an integer of type 'long long'
- **%llu**, **%Lu**: an integer of type 'unsigned long long'
- **%p**: the pointer of opaque data.
- **%x**: hexadecimal format of type 'int' in small alphabets
- **%X**: hexadecimal format of type 'int' in capital letters.
- **%lx**: hexadecimal format of type 'long' in small alphabets
- **%lX**: hexadecimal format of type 'long' in capital letters.
- **%Lx**: hexadecimal format of type 'long long' in small alphabets
- **%LX**: hexadecimal format of type 'long long' in capital letters
- **%le,%lg**: a scientific notion of type 'double'
- **%ls**: an wide-string where each character is of type 'wchar_t'

NOTICE: In printing floating number using uls-printf functions, Numbers ain't rounded up while the printf functions in stdio library do. Therefore the number strings by uls printf functions may be very slightly different from those by standard functions. The merit of this policy is that all the (floating) number digits printed by uls printf function are precisely a part of the number. But the rounding-up

floating number should be supported in future.

The conversion specification %S has a different meaning according to the string encoding. In the encoding of wide chars, format string and its arguments corresponding to conversion specifiers '%s' are wide strings. In the MS MS-MBCS Encoding, it should be a multibytes string. In the utf8 encoding system, it should be an utf8 string.

Below is the table summarizing the corresponding argument for '%s and '%S' according to the OS and encodings.

	ConvSpec	fmtstr(MBCS)	fmtstr(wchar_t)
Linux	%s	utf8 string	Wide-string
	%S	Wide-string	Wide-string
Windows	%s	MS-MBCS string	Wide-string
	%S	Wide-string	Wide-string

There're 2 encoding environments, MBCS(utf8 or MS-MBCS and WSTR(string of wchar_t), in programming. Wide string is used as format string in the WSTR environment and multibytes-string in the MBCS environment.

- Use the conversion specification '%ws' to print an wide string in the MBCS environment.
- Use the conversion specification '%ws' or '%S' to print an wide string in the WSTR environment

You can use the TEXT-macros like _T(), TEXT(), or TEXT-types like LPTSTR, LPCTSTR as in Windows, to make encoding-compatible source code.

9.3.1. String Printf

♦ `uls_snprintf(buf, bufsiz, fmt, ...)`

This procedure makes a formatted string from format string 'fmt' and the arguments displayed by the eclipse. The 'buf' is the memory of size 'bufsiz' in which the formatted string is stored. If the string exceeds the capacity of 'buf', the part of the string is truncated.

buf : (String) The output string buffer for formatted string by 'fmt' and args.

bufsiz : (int) the capacity of 'buf'

fmt : (ConstString) A format string

RETURN : **(int)** The length of the formatted string. In multibytes encoding mode. It is the number of bytes. In the wide-string encoding mode, it is the number of

(wchar_t) characters.

♦ `uls_zprintf(csz, fmt, ...)`

This procedure makes a formatted string from the 'fmt' and the arguments from the eclipse. Unlike the uls_snprintf, the resultant string is stored infinite, limited only by the system memory size, string as it uses the data of type 'csz_str_t'. To use this, do the following steps.

```
csz_str_t csz_buf;
char *formatted_string;
int len;

csz_init(&csz_buf, -1);
len = uls_zprintf(&csz_buf, "A format string %d %s %f\n", i, str, x);
formatted_string = csz_text(&csz_buf);
// Use the formatted string.
csz_deinit(&csz_buf);
```

When csz_deinit() is called, the memory of 'formatted_string' will be released. If you want to use the memory after the deallocating call, use csz_export() instead of csz_text().

```
formatted_string = csz_export(&csz_buf, &len);
```

After working with the memory, don't forget to release it.

```
uls_mfree(formatted_string);
```

csz : The output string buffer for formatted string by 'fmt' and varargs (csz_str_t*)

fmt : format string (ConstString)

RETURN : **(int)** The length of the formatted string

9.3.2. File Printf

♦ `uls_fprintf(fp, fmt, ...)`

The formatted string by 'fmt' and the arguments is emitted to the file of 'fp'

fp : output port. (FilePointer)

fmt : format string. (ConstString)

RETURN : **(int)** The length, # of bytes, of formatted utf8-string.

♦ `uls_printf(fmt, ...)`

The formatted string by 'fmt' and the arguments is emitted to the stdout.

fmt : format string.

(ConstString)

RETURN : **(int)** The length, # of bytes, of formatted utf8-string.

9.3.3. Generic Print

♦ `uls_sysprn_open(data, proc)`

This sets the internal output port for uls_vprint(). This registers 'data', 'proc' to the internal printing port. The data is just output port which is passed to the parameter of 'proc'.

data : output port.
(OpaqueData)

proc : A interface for putting (literal) string to output port.
(uls_lf_puts_t)

♦ `uls_sysprn_close()`

This declares that printing job to the internal output port is ended.

RETURN : **(int)**

♦ `uls_vsysprn(fmt, args)`

The formatted string by 'fmt' and the variable arguments is emitted to the output-port.
It can be used to your own printing function by wrapping this procedure as shown below.

```
int print(const char* fmt, ...) {
    va_list    args;
    int len;

    va_start(args, fmt);
    len = uls_vsysprn(fmt, args);
    va_end(args);

    return len;
}
```

fmt : A format string given by user
(ConstString)

args : A variable argument list
(VARARGS)

RETURN : **(int)** # of bytes written

♦ `uls_sysprn(fmt, ...)`

This procedure calls directly uls_sysprn().

fmt : A format string given by user (ConstString)

RETURN : **(int)** # of bytes written

9.3.4. Error Logging

♦ `err_log_puts(mesg)`

The log API having prefix 'err_' is for generic purpose. err_puts() simply emits the literal string 'mesg' to the internal output port. The internal error port is by default stderr and can be changed by err_change_port().

mesg : A message string having no format characters. (ConstString)

♦ `err_panic_puts(mesg)`

This prints a log message 'mesg' as in err_log_puts but the program will be aborted. Call this when it occurs, in runtime, an event that cannot be restored.

mesg : A literal string. (ConstString)

♦ `err_log(fmt, args)`

This emits the formatted string by 'fmt' and 'args' to the error port. The output is by default the 'stderr'. No need to append '₩n' to the end of the format string 'fmt'. You can use %t %w %k to print the status of uls lexical object.

- %t: To print the name of the current token. It requires the argument of type (uls_lex_t *) for it.
- %k: To print the keyword of the current token. It requires the argument of type (uls_lex_t *) for it.
- %w: To print the coordinate of the current input file or source. The coordinate is composed of the tag and line number. It requires the argument of type (uls_lex_t *) for it.

fmt : A format string given by user (ConstString)

♦ `err_panic(fmt, ...)`

This prints a log message using err_log() and the program will be aborted. Call this when it occurs an event that cannot be restored in runtime.

fmt : A format string given by user

(ConstString)

♦ `err_change_port(uls, data, proc)`

This sets the internal error port('data'), which is used by the logging API, err_log() and err_panic(). If the 'proc' is NULL, 'data' is a pointer of FILE, i.e., the output port will be regular file. The 'proc' is defined to the procedure that literal string is emitted to 'data'.

data : given by user
(OpaqueData)

proc : This argument of err_change_port() must have the following signature.
(uls_lf_puts_t)

```
int uls_lf_puts(void* dat, const char* str, int len)
```

The parameter 'dat' is the one that you gave in err_change_port(). The 'str' is the string to be printed to the 'dat' and the 'len' is the length of 'str'. The 'str' needn't to be null-terminated string.

9.3.5. ULS logging object

♦ `uls_create_log(lf_map, uls)`

This creates the main logging object supported by ULS-LF. The parameter 'lf_map' is a collection of conversion specifications. It's sufficient to pass NULL-pointer to make a default mapping. The 'uls' is the source object from which the log messages come.

lf_map : The structure having a mapping from conversion specification to it procedure.
(uls_lf_map_t*)

RETURN : **(uls_log_t*)** The ULS log object

♦ `uls_destroy_log(log)`

The counterpart of uls_create_log() to deallocate the memory of 'log'.

log : .the target structure to be destroyed.
(uls_log_t*)

♦ `uls_log(uls, fmt, ...)`

uls_log() is one of main API of the ULS logging framework. This emits the formatted string by 'fmt' and 'args' to the error port. The output is by default the 'stderr'. No need to append '₩n' to the end of the format string 'fmt'. It's capable of printing the keyword string of token and input coordinate, as well as default conversion specifications.

- %t: To print the name of the current token.
- %k: To print the keyword of the current token.
- %w: To print the coordinate of the current input file or source. The coordinate is composed

of the tag and line number.

There's no need to give the argument for the conversion specification as the 'log' object has already it. The object 'log' is associated with a 'uls' when it's created.'

fmt : A format string given by user
(ConstString)

♦ `uls_panic(uls, fmt, ...)`

This procedure logs an error message using uls_log() and the program will be aborted. Call this when it occurs, in runtime, an event that cannot be restored.

fmt : A format string given by user
(ConstString)

♦ `uls_log_change(log, data, proc)`

This redirects the error port of 'log'. The error port of uls_log_t is the target object to which error messages emit. By default, the error port is 'stderr' which is defined in the 'standard IO' library known as stdio. The uls_log_change() is used to change the error port with its procedure. The 'proc' is the procedure to send messages to the error port.

data : The error port given by user.
(OpaqueData)

proc : The procedure to manipulate the error port given by user.
(uls_lf_puts_t)

9.4. ULS utility procedures

♦ `uls_explode_str(ptr_line, delim_ch, args, n_args)`

This procedure separates a string pointed by 'ptr_line' as a word list and returns the list to the output parameter 'args'. The number of words is returned. The parameter 'n_args' is the capacity of 'args'. The number of words is excess of the capacity. The procedure will return the curtailed list. The parameter 'delim_ch' represents the delimeter characters for separating the string. For instance, delim_ch can be ' ', ',', ':', ';' and so on. As a special case, If the 'delim_ch' is specified as ' ', the procedure will consider the delimeter character set as ' ' and '₩t'.

Here is an example of uls_explode_str().

```
char *args[16], *line;
int n_words;
char *linebuff[81];

strcpy(linebuff, “word1, word2 word3 … wordn");
line = linebuff;

n_words = uls_explode_str(&line, ‘ ‘, args, 16);
// print args[0], args[1] …, args[n_words-1]
```

Notice that the line is modified to insert the null-character to indicate the end of each word.

ptr_line : (String*) The string to be splited as word list. The string pointed by 'ptr_line' cannot be the read-only one.

delim_ch : (char) delimeter character. If it's the space ' ', it's considered the tab character too.

args : (Array of String) OUT-PARAM The resultant word list is returned here.

n_args : (int) The capacity of the 'args'. It's up to user how many 'args[]' is needed.

RETURN : **(int)** the number of words separated.

♦ `splitstr(p_line)`

This separates words from a string. For example. The delimeter characters is ' ' and '₩t'. For example, To separate a word for a line line="A line string", use the procedure as follows.

```
word = splitstr(&line);
```

Then it will insert the line to null-character '₩0' and return the first word 'A' of 'line'. The procedure can be called again with p_line as 'p_line' is modified to point the next character of the first word. So, sequential call of 'splitstr' will split the string 'line'. The last call of the

procedure will return the null-string "''.

p_line : The string to be splited as word list
(String*)
INOUT-
PARAM

RETURN : **(String)** A word separate by space characters ' ' and '₩t'.

♦ `uls_filename(filepath, len_fname)`

This extracts the basename, the last component of the 'filepath', from 'filepath' and return the pointer of it in 'filepath'. If the parameter 'len_fname' is given, the procedure will retrieve the length that the length of its suffix is subtracted from the length of basename. If there's no suffix of the basename, the *len_fname will be zero. If you don't care the length, pass NULL-pointer to it. For instance, let the 'filepath' be '/topdir/repo/sample.ulc'.

```
fname = uls_filename(filepath, &len);
```

Then fname will be 'sample.ulc' with len == 5.

filepath : The input string as a file path
(ConstString)

len_fname : The basename of a filepath is the part of of it.
(int*) The length that the length of it suffix is subtracted from the length of basename
OUT-PARAM

RETURN : **(String)** The base name of the 'filepath'

♦ `uls_getopts(n_args, args, optfmt, opt_proc)`

It's similar to getopt() in glibc but used for the portability of ULS. It gets the option and its argument from the arguments 'args' by applying the procedure 'opt_proc' to each 'args'.
The 'optfmt' is the option format string. The 'opt_proc' should be a function having the following prototype.

```
int uls_optproc_t(int opt, char* optarg)
```

The parameter 'opt' is one of the character of 'optfmt' string. The 'optarg' is the argument of 'opt' if it is required. If the opt_proc() returns nonzero integer, the process of parsing argument is stopped and uls_getopts will return. If the value is negative, an error message is shown. If the value is positive, the procedure uls_getopts() is normally terminated.

n_args : The number of arguments of strings.
(int)

args : The arguments of string to be parsed.
(Array of String)

optfmt : (ConstString) The array of option characters. An additional option argument is required for th semi-colon attached character.

opt_proc : (uls_optproc_t) The sub procedure to process the pair of option character and its argument.

```
int uls_optproc_t(int opt, char* optarg)
```

If the option don't require the additional argument, don't use the second parameter 'optarg'.

RETURN : **(int)** the index of the non option argument in 'args[]'. Notice option argument must start with the minus sign '-'. If this value is not positive, it means that some errors have been occurred.

10. Class Library API

10.1. C++

10.1.1. UlsLex

- `UlsLex(ulc_file)`

  This is a constructor that creates an object for lexical analysis. The 'ulc_file' is a name of language specification in the ulc repository or simply a file path of ulc file. To see the available ulc names, use the –q-option of the command 'ulc2class'.

  ulc_file : The name/path of the lexical configuration
  (string)

- `LineNum`

  This members stands for the processing location of the current input. These can be read directly as they're declared as public members. As might be expected, the pair is updated as the input file is consumed.

- `setLineNum(lineno)`

  The field 'LineNum' is automatically updated by calling getTok() but if you want to change it forcibly use this method.

  lineno : The new value of 'LineNum' to be updated
  (int)

- `addLineNum(amount)`

  Use this method to add some lines to the current line number 'LineNum' forcibly. If the resultant line number is negative the 'LineNum' won't be updated.

  amount : The amount of lines to be added. It may be negative.
  (int)

- `getTag(tagstr)`
- `setTag(tagstr)`
- `getFileName(tagstr)`
- `setFileName(tagstr)`

  These methods is to get/set the processing location of the current input. As might be expected, The tag is updated as the input file is consumed.

- `pushLine(line, len)`

This method will push a literal string 'line' as an input source on the top of the input stack

line : A input source as an literal string
(char*)
len : The length of 'line'. The 'line' needn't to be null-terminated. If it's so, you may
(int) set len to -1.

♦ pushInput(fd)

This makes file descriptor 'fd' to be prepared on the internal stack. Processing of the previous input is postponed until the completion of processing the input from 'fd'.

fd : the file descriptor of input file
(int)
RETURN : **(bool)**

♦ pushInput(istr)

This method will push an input string 'istr' on the top of input stack. Then the getTok() method is used to get the tokens from the input.

istr : input stream object. You can create the input object from file(text or uls-file),
(UlsIStream) literal string.
RETURN : **(bool)**

♦ popInput()
♦ popAllInputs()

popInput() dismisses the current input source.

popAllInputs() dismisses all the input sources and goes back to the initial state. In the initial state you will get the EOI as current token.

♦ peekCh(isQuote)
♦ peekCh()

peekCh() peeks the next character in the input. getCh() will get the character and advance the cursor of input

isQuote : If the next token is a literal string this 'isQuote' is true.
(bool*)
OUT-PARAM
RETURN : **(uls_uch_t)** The next character

♦ getCh(isQuote)
♦ getCh()

This extracts the next character.

isQuote : (bool*) OUT-PARAM — If the returned value is UCH_UCH_NONE, check the value of 'isQuote' if the current toke is a literal string or not.

RETURN : **(uls_uch_t)** The next character

♦ `ungetCh(uch)`

This will push back the character 'uch' in order to get it again at the next call of peekCh() or getCh(). It will get the character advancing the cursor of input.

uch : (uls_uch_t)

♦ `getTok()`

This is one of the main methods of the uls lexical analyzer object. It gets a token and its associated lexeme advancing the cursor of input. At the end of input you will get a special token EOI. The EOI token number should be compared with the field 'toknum_EOI' as it's a dynamic number varied for each lexical configuration The procedure getTok() it returns the token number, you can get it also from the object. Use getTokNum() to get the current token number and getTokStr() to get the associated lexeme.

RETURN : **(int)** the token number

SYNONYM : next, getToken

♦ `getTokStr(ptr_str)`

It returns the current token string stored in the object by getTok(). ptr_str is the pointer of tstring, which may be string or wstring according the used character set. The returned value ptr_str is pointer of tstring. To get the token string use ptr_str->c_str(). Don't modify the content of ptr_str directly. The value is valid before the next call of getTok().

RETURN : **none**

♦ `isLexemeInt()`
♦ `isLexemeReal()`
♦ `isLexemeZero()`

These methods check if the lexeme of the current token is an integer or floating-point number, or zero.

RETURN : **(bool)**

♦ `lexemeAsInt()`

lexemeAsInt() will recognize the current token string as an integer. It returns the integer value after converting the token string to primitive type 'int'. Make sure the current token is a number.

RETURN : (int)

♦ `lexemeAsUInt()`

lexemeAsUInt() will recognize the current token string as an integer. It returns the integer value after converting the token string to primitive type 'unsigned int'. Make sure the current token is a number.

RETURN : (unsigned int)

♦ `lexemeAsLongLong()`

lexemeAsLongLong() will recognize the current lexeme as an 'long long'. It returns the integer value after converting the token string to primitive type 'long long'. Make sure the current token is a number.

RETURN : (long long)

♦ `lexemeAsULongLong()`

lexemeAsULongLong() will recognize the current lexeme as an 'unsigned long long'. Make sure the current token is a number.

RETURN : (unsigned long long)

♦ `lexemeAsDouble()`

lexemeAsDouble() will recognize the current lexeme as a 'double'. It returns the double floating number after converting the token string to primitive type 'double'. Make sure the current token is a number.

RETURN : (double)

♦ `lexemeNumberSuffix()`

In case that the current token is NUMBER, it'll return the suffix of number if it exists. It is obviously used to inform compiler of its data types in programming language.

RETURN : (string) suffix string

♦ `expect(TokExpected)`

If the current token-id is not TokExpected, An exception will be thrown.

TokExpected : It's expected that the current token number is equal to 'TokExpected'. Otherwise
(int) the program will be aborted throwing an exception.

- `setExtraTokdef(t, extra_tokdef)`

This sets extra token definition 'extra_tokdef' as opaque data, which should be provided by user. The stored data of token number 't' can be later used to retrieve the 'extra_tokdef'. Refer to the example in tests/dump_toks for example.

t : The target token number with which user data is associated.
(`int`)

extra_tokdef : The extra tokdef supplied by user.
(void*)

- `getExtraTokdef()`
- `getExtraTokdef(t)`

Use this method to get the user-defined token information stored previously by setExtraTokdef(). Refer to the example in tests/dump_toks for example.

t : The token id of data the you want to retrieve.
(`int`)

RETURN : **(OpaqueData)** The extra tokdef is provided by user.

- `ungetTok()`
- `ungetStr(lxm)`
- `ungetLexeme(lxm, tok_id)`

Call ungetTok() if you want to get the current token again after getTok().
Call ungetStr() if you want to push in a string to the current input source.
Call ungetLexeme() if you want to push in lexeme to the current input source.

lxm : The token string with which the token number 'tok_id' is paired.
(`string`)

tok_id : token number
(int)

- `dumpTok(pfx, suff)`
- `dumpTok()`

This dumps the current token as explanatory string, which is composed of the string 'pfx', 'suff', and the information of the token. For example, if pfx is '₩t' and suff is '₩n', the output line in the terminal may be like as follows.

'₩t' [__ID] main '₩n'

pfx : The prefix string of the line

(string)

suff : The suffix string of the line

(string)

♦ `getKeywordStr(ptr_keyw)`

Can get the keyword string corresponding to the token number 't' via ptr_keyw->c_str(). For example, if the token number is 1 and it represents the keyword '+=', call the method getKeywordStr(1, ptr_keyw) ptr_keyw->c_str() should return the literal string '+='.

t : The token number of which keyword you want to know.

(int)

RETURN : (string)

SYNONYM : Keyword

10.1.2. UlsTmplList

♦ `UlsTmplList(size)`

UlsTmplList is the class to store a list of template variables with their values. It's passed to the argument of UlsIStream(). A template variable is composed of a name and its (string) value.

size : The capacity of the list.

(int)

♦ `clear()`

This clears the internal list of variables and makes the length of the list zero.

♦ `exist(tnam)`

It checks if the variable 'tnam' exists in the list.

tnam :

(string)

RETURN : (bool)

♦ `length()`

It returns the length of the internal list. It's always less than or equal to the capacity of the list.

RETURN : (int) # of variables.

♦ `getValue(tnam, tval)`

It returns the value of 'tnam'. If there's no template variable named 'tnam', it returns false. Otherwise the value of the 'tnam' is copied to 'tval' and it returns true.

tnam : the name of template variable
(string)
tval : the value of the 'tnam'
(string)
OUT-PARAM
RETURN : (bool)

♦ `setValue(tnam, tval)`

This modifies the pair <tnam, tval> in the internal list if the item named 'tnam' exists.

tnam : the name of template variable
(string)
tval : the value of the 'tnam'
(string)

♦ `exportTmpls(tmpllist_exp)`

This exports the internal list to another UlsTmplList object.

`tmpllist_exp` : A new UlsTmplList object to which the current list is exported.
(UlsTmplList)
OUT-PARAM
RETURN : (int) the length of the list

10.1.3. UlsIStream

♦ `UlsIStream(filepath, uls_tmpls)`

UlsIStream is used as an input file of UlsLex. It's an abstraction of text (program source) file or token sequence file 'uls'. The parameter 'uls_tmpls' is optional. In case that the file is a token sequence file(*.uls) and has template variables, the parameter 'uls_tmpls' is needed.

filepath : Input of UlsLex
(string)
uls_tmpls : A list of template variables having its values too.
(UlsTmplList*)

10.1.4. UlsOStream

♦ `UlsOStream(filepath, lex, subtag="", numbering=true)`

This makes an object that supports write-only sequential IO. The purpose is to save the token sequence form input source.

filepath : (string) A file path for the output uls-stream file. If the file already exists it'll be overwritten.

lex : (UlsLex*) The lexical analyzer associated with the uls-stream file

subtag : (string) An user-provided tag for the file 'filepath'. More often than not, it tends to be the file path. This string will be written to the output uls file.

numbering : (bool) Specify whether the number information is to be inserted or not. The information is automatically inserted whenever the line of source code is changed.

♦ `close()`

This finalizes the task of streaming. It flushes data buffer and closes the output-file.

♦ `printTok(tokid, tokstr)`

This method prints a record of <tokid, tokstr> pair to the output stream.

tokid : (int) the token number to be printed in the output file.

tokstr : (string) the lexeme associated with the 'tokid'

♦ `printTokLineNum(lno, tagstr)`

This method prints an annotation <linenum, tag> pair to the output file.

lno : (int) the line number of the source file

tagstr : (string) the tag of the source file.

♦ `startStream(ifile)`

It starts writing the lexical streaming with input-stream 'ifile'.

ifile : (UlsIStream) This input uls-stream will be written to the 'UlsOStream' object

RETURN : (bool)

10.1.5. Logging Framework

10.1.5.1.String Printf

♦ `snprintf(buf, bufsiz, fmt, ...)`

This procedure makes a formatted string from format string 'fmt' and variable arguments list. The 'buf' is a buffer of size 'bufsiz' where the formatted string is stored. In case of the resultant string being exceeded the capacity of the 'buf', the string will be truncated to the 'bufsiz'.

buf : The output buffer for the formatted string
(char *)
bufsiz : The capacity of 'buf'
(int)
fmt : format string
(char *)
RETURN : **(int)** # of chars filled

10.1.5.2.File Printf

♦ `fprintf(fp, fmt, ...)`

The formatted string by 'fmt' and the arguments is emitted to file.

fp : FILE pointer that is opened by fopen().
(FILE *)
fmt : format string
(char *)
RETURN : **(int)** # of chars filled

♦ `printf(fmt, ...)`

The formatted string by 'fmt' and the arguments is emitted to the stdout, standard output.

fmt : format string
(char *)

10.1.5.3.Generic Printf

♦ `openOutput(out_file)`

This opens a file for writing text messages sequentially. The control data of the file is internally managed by the system. To write to the output file, use print() defined below.

`out_file` : The output file path
(string)

♦ `closeOutput()`

This flushes and closes the buffer of the output. After using the calls of print() method, be sure to call closeOutput().

♦ `print(fmt, ...)`

This will print the formatted string to the output opened by openOutput().

fmt : format string
(char *)
RETURN : **(int)** # of chars printed

10.1.5.4.Error Logging

♦ `log(fmt, ...)`

This is one of main methods that ULS logging framework gives. This emits the formatted string from 'fmt' and 'args' to the error port. The output is by default the 'stderr'. No need to append '₩n' to the end of the format string 'fmt'. It is capable of printing the keyword string of token and input coordinate, as well as the default conversion specifications.

- %t: To print the name of the current token.
- %k: To print the keyword string of the current token.
- %w: To print the coordinate of the current input file or source. The coordinate is composed of the tag and line number.

Notice that there's no need to give the argument for the conversion specification as the 'log' object has already a reference of it. The object 'log' is associated with a 'uls' when it's created.'

fmt : format string
(char *)

♦ `panic(fmt, ...)`

This informs user of the occurrence of a system error with a formatted message. The program will be aborted.

fmt : The template for message string
(char *)

10.2. C# and Java

Below is listed the common API of C# and Java. C# or Java specific procedure is stated explicitly.

10.2.1. UlsLex

♦ `UlsLex(ulc_file)`

This is a constructor that creates an object for lexical analysis. The 'ulc_file' is a name of language specification in the ulc system repository or simply a file path of ulc file. To see the available ulc names, use the –q-option of the command 'ulc2class'.

ulc_file : The name/path of the lexical configuration
(String)

♦ `Tag`

[C# Only] This stands for the processing location of the current input and can be read directly as declared as public property. This is updated as the input file is read.
In Java, To get/set the 'Tag', use getTag() or setTag(tag) method.

♦ `LineNum`

[C# Only] This stands for the processing location of the current inpu and can be read directly as declared as public property. This is automatically updated as the input file is read.
In Java,
to get/set the 'LineNum', use getLineNum(), setLineNum(lineno), or AddLineNum(amount).

♦ `pushInput(line)`

This method will push the literal string as an input source on the top of the stack

line : An input source as an literal string.
(`String`)

♦ `pushInput(filepath)`

This makes an input-file to be ready by putting it on the internal stack of input.

filepath : The file path of input
(`String`)
RETURN : (bool)

♦ `pushInput(istr)`

This method pushes an input string 'istr' on the top of input stack. Then the getTok() method can be used to get the tokens from the input.

istr : (`UlsIStream`) The input stream object. You can create this input object from file(text or uls-file), literal string.

RETURN : (bool)

- `popInput()`
- `popAllInputs()`

popInput() dismisses the current input source.

popAllInputs() dismisses all the input sources and goes back to the initial state. In the initial state you will get the EOI as current token.

- `peekCh(uch_detail)`

peekCh() peeks the next character in the input. getCh() will get the character and advance the cursor of input.

uch_detail (GetchDetailInfo) OPTIONAL: It can't break literal string to characters. This procedure run into a literal string, its information is returned in the parameter 'uch_detail'.

- ✓ uch_detail.tok_id : the token number of quotation string, EOF, EOI, or NONE.
- ✓ uch_detail.is_quote: true if the current position indicated the literal string false otherwise.

RETURN : (uls_uch_t) The next unicode codepoint

- `getCh(uch_detail)`

It extracts the next character.

uch_detail (GetchDetailInfo) OPTIONAL: It can't break literal string to characters. When this procedure comes into a literal string, the information of the string is returned in the parameter 'uch_detail'.

- ✓ uch_detail.tok_id : the token number if the current token is a quotation string, EOF, EOI, or NONE otherwise.
- ✓ uch_detail.is_quote: true if the current position indicated the literal string false otherwise.

In case the current token is a literal string, you can use the push-line interface to break the string into characters.

RETURN : (uls_uch_t) The next unicode codepoint

♦ `ungetCh(uch)`

This will push back the character 'uch' to get it again at the next call of peekCh() or getCh(). getCh() will get the character advancing the cursor of input.

uch :
(uls_uch_t)

♦ `getTok()`

This is one of the main methods of the uls lexical analyzer object. It gets a token and its associated lexeme advancing the cursor of input. At the end of input you will get a special token toknum_EOI. Even if it returns the token number, you can get it later also from the object. Use the property 'TokNum' to get the current token number and the 'TokStr' to get the associated lexeme.

RETURN : **(int)** the token number

SYNONYM : next, getToken

♦ `TokNum`

[C# Only] This is the current token number obtained by getTok(). The value is valid before the next call of getTok().

SYNONYM : token

♦ `TokStr`

[C# Only] This is the current token string stored in the object by getTok().
The value is valid before the next call of getTok().

SYSNONYM : lexeme

♦ `isLexemeInt()`
♦ `isLexemeReal()`
♦ `isLexemeZero()`

These methods check if the lexeme of the current token is integer or floating-point number, or zero.

RETURN : **(bool)**

♦ `lexemeAsInt(), lexemeAsLong()`

[Java Only] This procedure will recognize the current token string as an integer. It returns the integer value after converting the token string to primitive type 'int' or 'long'. Make sure the current token is a number.

RETURN : (int), (long)

♦ `lexemeAsInt32(), lexemeAsUInt32()`

These recognize the current token string as a 32-bit integer. It returns the integer value after converting the token string to primitive type 'Int32'. Make sure the current token is a number.
The lexemeAsUInt32 should be used in only C# as there's no unsigned primitive type in Java.

RETURN : Int32, UInt32

♦ `lexemeAsInt64(),lexemeAsUInt64()`

These will recognize the current lexeme as 64-bit integer. Make sure the current token is a number. The lexemeAsUInt64 should be used in only C# as there's no unsigned primitive type in Java.

RETURN : Int64, UInt64

♦ `lexemeAsDouble()`

lexemeAsDouble() will recognize the current lexeme as an 'double'. It returns the integer value after converting the token string to primitive type 'double'. Make sure the current token is a number.

RETURN : (double)

♦ `lexemeNumberSuffix()`

In case that the current token is NUMBER, this'll return the suffix of number if it exists.

RETURN : (String) suffix string

♦ `expect(TokExpected)`

If the current token id is not TokExpected, An exception will be thrown.

TokExpected : (int) It's expected that the current token number is equal to 'TokExpected'. Otherwise the program will be aborted throwing an exception.

♦ `setExtraTokdef(t, extra_tokdef)`

This sets extra token definition 'extra_tokdef' as opaque data which should be given by user. The stored data for token number 't' can be later retrieved by getExtraTokdef().

t : (`int`) The target token number with which user data is associated.

extra_tokdef : The extra tokdef provided by user.

(Object)

- `getExtraTokdef(t)`
- `getExtraTokdef()`
- `ExtraTokdef`

Use this method to get the user-defined token information stored previously by setExtraTokdef().

t : target token id of data the you want to retrieve..

(`int`)

RETURN : **(OpaqueData)** The extra tokdef is provided by user.

- `ungetTok()`
- `ungetStr(lxm)`
- `ungetLexeme(lxm, tok_id)`

Call ungetTok() if you want to get the current token again after getTok().

Call ungetStr() if you want to push in a string to the current input source.

Call ungetLexeme() if you want to push in lexeme to the current input source.

lxm : The token string with which the token number 'tok_id' is paired.

(`String`)

tok_id : token number

(int)

- `dumpTok(pfx, suff)`
- `dumpTok()`

This dumps the current token with explanatory string, which is composed of the string "pfx", "suff", and the information of the token. For example, if pfx is '₩t' and suff is '₩n', the output line to the terminal may be like as follows.

'₩t' [__ID] main '₩n'

pfx : The prefix string of the line.

(`String`)

suff : The suffix string of the line.

(`String`)

- `getKeywordStr(t)`
- `getKeywordStr()`

This returns the keyword string associated with token number 't'. For example, if the token number is 1 and it represents the keyword '+=', getKeywordStr(1) returns the string '+='.

t : The token number of which keyword you want know.

(`int`)

RETURN : **(string)**

SYNONYM : Keyword

10.2.2. UlsTmplList

♦ `UlsTmplList(size)`

UlsTmplList is a class to store a list of template variables with their values. It can be passed to the argument of UlsIStream(). A template variable is composed of a name and its (string) value.

size : The capacity of the list.
(`int`)

♦ `clear()`

This clears the internal list of variables and makes the length of the list to be zero.

♦ `exist(tnam)`

This checks if the variable 'tnam' exists in the list.

tnam :
(String)
RETURN : **(bool)**

♦ `length()`

It returns the length of the internal list, which is always less than or equal to the capacity of the list.

RETURN : **(int)** # of variables.

♦ `add(tnam, tval)`

Append a pair <tnam, tval> to the list increasing the length of the list.

tnam : the name of template variable
(String)
tval : the value of the 'tnam'
(String)

♦ `getValue(tnam, tval)`

It returns the value of 'tnam'. If there's no template variable named 'tnam', it returns 'null'.

tnam : The name of template variable
(String)
tval : The value of the 'tnam'
(String)
OUT-PARAM
RETURN : **(bool)**

♦ `setValue(tnam, tval)`

This modifies the pair <tnam, tval> in the internal list if the item named 'tnam' exists.

tnam : The name of template variable
(String)
tval : The value of the 'tnam'
(String)
RETURN : **(bool)**

♦ `exportTmpls(tmpllist_exp)`

This exports the internal list to another UlsTmplList object.

`tmpllist_exp` : A new UlsTmplList object to which the current list is exported.
(UlsTmplList)
OUT-PARAM
RETURN : **(int)** the length of the list

♦ `operator [tnam]`

Use mapping operator [] to look up the value of template variable. For example, To know the value of variable 'AAA' of UlsTmplList,

```
UlsTmplList  tlist;
String str = tlist["AAA"];
```

RETURN : **(String)** the value of 'tnam'

10.2.3. UlsIStream

♦ `UlsIStream(filepath, uls_tmpls = null)`

UlsIStream can be used as an input file of UlsLex. It's an input abstraction containing text (program source) file and token sequence file 'uls'. The parameter 'uls_tmpls' is optional. In case that the file is a token sequence file and has template variables, the parameter 'uls_tmpls' is needed.

filepath : Input of UlsLex
(String)

uls_tmpls : (UlsTmplList*) A list of template variables having its values too.

♦ `close()`

This finalizes the task of streaming and closes the output file. When the object is destroyed this method is called.

10.2.4. UlsOStream

♦ `UlsOStream(filepath, lex, subtag="", numbering=true)`

This makes the object that supports for write-only sequential IO. Its purpose is to save the token sequence from input source.

filepath : (String) It's a file path for a new uls-stream file. If the file already exists it'll be overwritten.

lex : (UlsLex*) The lexical analyzer associated with the uls-stream file

subtag : (String) A user provided tag to the file 'filepath'. More often than not, it tends to be the file path. This string will be written to the output uls file.

numbering : (bool) Specify whether the number information is to be inserted or not. The information is automatically inserted whenever the line of source code is changed.

♦ `close()`

This finalizes the task of streaming. It flushes data buffer and closes the output-file.

♦ `printTok(tokid, tokstr)`

This method prints a record of <tokid, tokstr> pair to the output stream.

tokid : (int) token id to be printed

tokstr : (String) the lexeme associated with the 'tokid'

♦ `printTokLineNum(lno, tagstr)`

This method prints an annotation <linenum, tag> pair to the output file.

lno : (int) the line number of the source file

tagstr : the tag of the source file.

(String)

♦ `startStream(ifile)`

Start writing the lexical streaming with input-stream 'ifile'.

ifile : This input uls-stream will be written to the 'UlsOStream' object (UlsIStream)

RETURN : (bool)