



HORION s.n.c. di L. Mini & C.

[www.horion.it](http://www.horion.it)

[info@horion.it](mailto:info@horion.it)

Tel.: +39 011 19825680

skype: horion\_automation

---

## **HLang - Another But Very Simple Parser**

**Version 1.0**

rev.: 01/07/2009

This is a simple class written in C + + standard for building parsers but does not generate a source to be completed by the user, but it is interpreted run-time.

The class consists of the following files:

The class itself:

Hlang.h

Hlang.cpp

other routines used by the class Hlang:

Tokenizer.h

Tokenizer.cpp

Trace.h

Trace.cpp

Utils.h

Utils.cpp

in addition to the file that describes the grammar that can have any name.

The Trace class has only function debug, showing messages to stdout.

The class is not optimized and can certainly be made improvements, but given the scope of this class is not a problem.

## **Features**

- scanner included
- recursive parser LL(k)
- automatic building of the syntactic tree
- grammar interpreter run-time
- can easily handle multiple parsers with different grammars
- semantic implementation is not intrusive in the grammar file specification to facilitate debugging
- support for executing the user program
- multi-pass support
- it uses STL and regex libraries.

Many functions such as the scanner, the function to run the program, etc.. are declared virtual so that in case of special needs can be easily redefined.

The logical flow of operation is as follows:

1. load the grammar specification
2. load the user program
3. parsing
4. executing

The execution is done by performing the various functions in separate moments of the interpretation itself . Infact, in order to achieve good performance in speed of execution as the program is first interpreted and then executed. The overhead for each function is very small. However you can separate the process of parsing from the stage of execution.

## Usage

Let's see first some basics and then some simple examples of use. In this regard, it should examine the example code provided.

Public members of Hlang\_cls class are:

- `int Hlang_cls::ParseLang(bool exe)`
- `virtual bool Hlang_cls::LoadProgram(string fname)`
- `virtual string Hlang_cls::GetProgramToken(string::iterator token_req)`
- `virtual bool Hlang_cls::MacroTerminals(string &symb, string &token)`
- `virtual void Hlang_cls::ExeProgram(void)`
- `void Hlang_cls::Clear()`
- `void Hlang_cls::ClearAll()`
- `virtual int ExeFuncDispatch(int pos, int func, vector<string> &params, vector<int> &paramstype)`

First we must derive the class so that members have access to virtual especially for the management of semantics.

The most important function is the one responsible for managing the semantics, the `ExeFuncDispatch` that must implement all the custom functions that give meaning to the various parts of the program described in the grammar

## Hlang Class usage

When the class is allocated, the constructor receives a file name, the file that contains the grammar description. For example, the constructor of the class `example_cls` will be:

```
example_cls::example_cls():Hlang_cls("grammar.g")
{
}
```

When the class will be allocated `example_cls` the manufacturer `Hlang_cls` load grammar described in the file `grammar.g`.

## Membro per le funzioni semantiche

We must then implement the function `ExeFuncDispatch` that will be called once interpreted the user program. This function has the following parameters:

- |                   |  |
|-------------------|--|
| <code>pos</code>  | is the number of the logical position of the interpreted instruction   |
| <code>func</code> | every function associated with the semantics of a given symbol is represented by a positive integer (see also the <code>grammar.g</code> )<br>There are two functions that are always executed: <ul style="list-style-type: none"><li>- 1: before run the user program, usefull for init</li><li>- 2: at the end of the user program</li></ul> |

params	parameters are passed to the function as a vector of strings. It's a task of the user functions to convert it as appropriate.
paramstype	parameters type as specified by the grammar. it's a vector of integers.

An easy way to implement the semantic functions is as follows:

```
int example_cls::ExeFuncDispatch(int pos, int func, vector<string>
&params,vector<int> &paramstype)
{
switch(func)
{
case -1: // init
// todo
break;
case -2: // end
// todo
break;

case 1:
// todo
break;
case 2:
// todo
break;

}
return (HLANG_RET_NEXT);
}
```

This member must return an integer that will have the following meanings depending on its value:

If the value is greater than or equal to 0 indicates the next instruction to be executed, this is useful if you need to implement instructions such "for", "while" or "goto" that need to be able to jump to a particular point of the program.

If the value returned is less than 0, acquires special significance:

HLANG\_RET\_NEXT

execute the natural next instruction as result from the program

HLANG\_RET\_EXIT

stop and exit from the execution of the program

HLANG\_RET\_DEC\_LEVEL

decrease the nesting level of a block (see later)

HLANG\_RET\_INC\_LEVEL

increase the nesting level of a block (see later)

HLANG\_RET\_ERROR

exit with error

HLANG\_RET\_NEXTPASS

in case of implementation of languages that require more passes (multiple

executions), indicate when is the case to get the next pass. The first is 1 and the current pass is stored in the public property called `pass`

NOTE: At every pass, the program is executed from the beginning and the first function executed is the - 1

`HLANG_RET_NEXTIGNORELEVEL`

execute the next instruction regardless the nesting level of the current block

If you need to handle a multi-pass execution:

```
int example_cls::ExeFuncDispatch(int pos, int func, vector<string>
&params,vector<int> &paramstype)
{
if (pass==1)
    {
    switch (func)
        {
        case -1:    // init
            // todo
            break;
        case -2:    // end
            // todo
            break;

        case 1:
            // todo
            break;
        case 2:
            // todo
            break;
        }
    }
else
    {
    switch (func)
        {
        case -1:    // init
            // todo
            break;
        case -2:    // end
            // todo
            break;

        case 1:
            // todo
            break;
        case 2:
            // todo
            break;
        }
    }
return (HLANG_RET_NEXT);
}
```

By this function, you can alter the functioning of the system to run the program. There are properties that can be read / changed:

pass

specify the current pass.

next\_pass

set to true, specify that you want to do another pass of the program. This function exists as return value too, but this allow you to alter the functioning and return another value if you need.

ignorelevel\_lock

allows you to override the nesting level ignoring what this function returns. For example, if you are looking for label inside the program for the jumps with a goto like function, it is useful to ignore the blocks, so the level should not be altered, because the label is absolute and not relative to the block where the label is.

stop\_script

if it is true, stop the execution

## **A look to the scanner**

The class implements a scanner that allows with no effort to retrieve tokens from the program. This member is virtual so it can be implemented according to your own needs, but for many applications should be enough, as provided here.

The scanner get the token specified in the grammar, both terminals and macroterminals. It scan all the program in search of the match that generates the longer token. Tokens in the productions have the priority, and then which one specified at the beginning of the file specified as regular expressions.

Its operation is quite similar to "flex" of course with the limitations of the case.

## ***Descrizione della grammatica***

The grammar of the language to be implemented has to be described in a text file. Below is an example:

```
&comment_line           //
&comment_block_start   /*
&comment_block_end     */

# terminal symbol %T
&terminal 1 %Tnumb_i    ^[-+]?[0-9]+$
&terminal 2 %Tvar       ^[a-zA-Z_][a-zA-Z_0-9]*$
&terminal 3 %Tnumb_hex  ^[a-fA-F0-9]+$
&terminal 4 %Tans       ANS
&terminal 5 %Tans[     ANS\[
&terminal 6 %Tlabel     ^[a-zA-Z_][a-zA-Z_0-9]*[\ \t]?:$
&terminal 7 %Talphanum  ^\w+$
&terminal 8 %Tchan      ^\w+\.[0-9]+$
&terminal 9 %Tstring    ^"(\\"|"[^"])*"$
```

```

TOP := function %Tvar %block                                $F200 ( )
block := %block_init %body %block_end
block_init := {                                           $+      $F100 ( )
block_end := }                                           $*-     $F101 ( )

body := %statement %body
body :=

statement := %var_assign
statement := %if_stat
statement := %repeat_stat
statement := %sample_stat
statement := waitms %Tnumb_i                               $F7 ( 2 )
statement := %send
statement := exit                                         $F15 ( )
statement := goto %Tvar                                   $F19 ( 2 )
statement := %Tlabel                                     $F20 ( 1 )

var_assign := %Tvar = %Tans                                $F1 ( 1 )
var_assign := %Tvar = %Tans[ %Tnumb_i ]                  $F2 ( 1 4 )
var_assign := %Tvar = %Tnumb_i                           $F3 ( 1 3 )
var_assign := %Tvar = %bexpr                              $F1 ( 1 )
var_assign := %Tvar = [ %arrayval ]                      $F1 ( 1 )

arrayval := %Tnumb_i %arrayval                            $F18 ( 1 )
arrayval :=

if_stat := if ( %bexpr ) then %block                      $F4 ( 6 )
repeat_stat := repeat %Tnumb_i %block                     $F6 ( 2 3 )

# simple expressions
bexpr := %Talphanum == %Talphanum                       $F10 ( 1 2 3 )
bexpr := %Talphanum != %Talphanum                       $F10 ( 1 2 3 )
bexpr := %Talphanum < %Talphanum                        $F10 ( 1 2 3 )
bexpr := %Talphanum > %Talphanum                        $F10 ( 1 2 3 )
bexpr := %Talphanum <= %Talphanum                      $F10 ( 1 2 3 )
bexpr := %Talphanum >= %Talphanum                      $F10 ( 1 2 3 )
bexpr := %Talphanum + %Talphanum                       $F10 ( 1 2 3 )
bexpr := %Talphanum - %Talphanum                       $F10 ( 1 2 3 )
bexpr := %Talphanum * %Talphanum                      $F10 ( 1 2 3 )
bexpr := %Talphanum / %Talphanum                      $F10 ( 1 2 3 )
bexpr := %Talphanum << %Talphanum                     $F10 ( 1 2 3 )
bexpr := %Talphanum >> %Talphanum                     $F10 ( 1 2 3 )

send := send %Tstring                                    $F14 ( 2 )

```

see in detail the meaning of all this.

At the moment it isn't very important what kind of grammar is to describe.

Comments are allowed, but only of whole lines and starting with "#".

At the beginning of the file are specified some definitions of internal variables that are useful for simplifying the work of the user. The internal variables, starting with "&", are:

`&comment_line str`

allows you to specify which symbol in the program is used to define lines of comments, like "\\\" in C++ for example. These lines are ignored in the process of loading the program.

`&comment_block_start str`

allows you to choose the symbol of the starting blocks for comment, like "/\*" in C programs. The content of these blocks will be ignored in the process of loading the program.

`&comment_block_end str`

allows you to choose the symbol of the end for the blocks of comments, such as "\*/" in C programs. The content of these blocks will be ignored in the process of loading the program.

`&terminal ntype %Tname regex`

Specify a terminal symbol by a regular expression. These symbols are hereinafter called "macroterminals". You must specify three parameters:

<code>ntype</code>	integer number in the range 1..999. It's the value passed in the <code>paramstype</code>
<code>%Tname</code>	symbol name as is used in the grammar productions. It must begin with %T
<code>regex</code>	regular expression. Its syntax is the same as specified in the <code>regex.h</code> library. Please refer to it for the details.

`&delimiters [str]`

Allows you to specify which separators for tokens are used in the program. If not specified, the default is the string "\\t\\n\\r". Note that the brackets "[" and "]" are necessary

The terminal symbols used, are taken directly from the description of the grammar productions, so we don't have to specify nothing for them.

The non terminal symbols must start by "%" when used in the production, which should not follow a "T", which would otherwise be confused with the macroterminal symbols. The % symbol must not be placed instead in the definitions of non-terminal symbol.

The production will have this format:

`non_terminal_symb := production semantic`

If there are productions in "OR" you have to repeat the non-terminal symbol and describe the production as an alternative. for example take a look at the non-terminal symbol "statement".

### **Semantic**

The simplest is the empty line!

Each production, may have only one function associated semantics, specified by the symbol  $\$Fn$ , where  $n$  is a positive integer. For example, we note the following line:

```
statement := exit                                $F15 ()
```

when the parser identifies this production, the function responsible for the execution will call the member `ExeFuncDispatch` to run the function 15 without any parameter, specified by `()` with no spaces between the two brackets. Let the following line

```
bexpr := %Talphnum == %Talphnum                $F10 ( 1 2 3 )
```

In this case will be called the function 10, but this time will be passed three parameters. The number in parentheses indicates the position of the symbol in the grammar (please pay attention to the spaces). Number 1 is the first symbol in the production, 2 the second and so on.

You can then have two cases:

1. the symbol refer to a terminal symbol or a macroterminal one, so the parameter passed to the function is the symbol itself
2. the symbol is a non terminal one, so the parameter is teh position of that symbol in the syntactic tree

in the above example, suppose that corresponds to one row of the program of this kind, of course, corresponds to this production

```
alpha == 23
```

we have these parameters: the first (position 1) will be `alpha`, the second (position 2) will be `==` and the third (position 3) will be `23`.

Take a look to another example:

```
if_stat := if ( %bexpr ) then %block           $F4 ( 6 )
```

As you note, the function 4 receives a parameter which corresponds to a non terminal symbol so it will be passed in the list of parameters the position of the first terminal symbol identified in the production, which describes the symbol `%block`.

In fact, the parser, as they analyze the program, in turn builds a database of all the terminals symbols found in order to build the syntactic tree. So, the value passed to the function semantics when the symbol is a non-terminal, you can see it as a pointer to a specified point of the syntactic tree.

The production above match a construct of the type

```
if ( alpha == 13 ) then
{
  alpha = 45      <- prima istruzione del blocco
}
```

that, in reference to the above, the parameter corresponds to the position 6 at runtime will be the position of the first instruction that is in the `%block` (in this

case alpha = 45).

There are some other peculiarities in semantics. suppose you want to implement a language that has the ability to nest constructs. To do this we use a variable that indicates the level of nesting so the function that runs the program knows which is the next instruction to be executed corresponding to the right level. For example:

```
block_init := {                               $+      $F100 ()
```

this production says that when you meet the "{" you must increase the level of nesting (done by means of \$+) and run the function 100 that has no parameters. The corresponding line that shows how you exit from a nested block is for example

```
block_end := }                               $*-     $F101 ()
```

in this case says that when you meet the "}" you should decrease the level of nesting (done by means of \$\*-) and call the function 101 that has no parameters. Note that the level can be increased/decreased before or after they have entered or left the block.

Using the symbols \$+ and \$- the level of indentation is increased/decreased in advance of entry/exit in/from the block, and with the symbols \$\*+ and \$\*- is increased/decreased after you have entered/exited.

In the example, the symbol "{" will be part of the next level and also the symbol "}" because we have indicated that decrease the level after the storage with \$\*-.

-- end --