



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

www.horion.it

info@horion.it

Tel.: +39 011 19825680

skype: horion_automation

HLang - Another But Very Simple Parser

Versione 1.0

rev.: 07/01/2009

Si tratta di una semplice classe scritta in C++ standard per costruire parser che però non genera un sorgente da compilare con il programma utente, bensì viene interpretato run-time.

La classe è costituita dai seguenti files:

La classe vera e propria:

Hlang.h

Hlang.cpp

altre routine utilizzate dalla classe Hlang:

Tokenizer.h

Tokenizer.cpp

Trace.h

Trace.cpp

Utils.h

Utils.cpp

oltre al file che descrive la grammatica che può avere nome qualsiasi.

La classe Trace ha solo funzione di debug, mostrando sullo stdout i messaggi.

La classe non è ottimizzata e possono essere fatte certamente migliorie in tal senso, ma visto lo scopo di questa classe, non risulta essere un problema.

Caratteristiche

- tokenizer incluso
- parser ricorsivo LL(k)
- costruzione dell'albero sintattico automatica
- interprete della grammatica run-time
- possibilità di coesistenza di molti parser con grammatiche differenti
- semantica implementata interamente nel programma utente, in modo da facilitarne il debug
- funzione per l'esecuzione del programma utente
- implementazione di parser e compilatori multi passata
- utilizza le STL e la libreria regex.

Molte funzioni come il tokenizer, la funzione per l'esecuzione del programma ecc. sono dichiarate virtuali cosicché in caso di esigenze particolari è possibile ridefinirle.

Il flusso logico di funzionamento è il seguente:

1. caricamento della grammatica
2. caricamento del programma utente (scritto nel linguaggio definito dalla grammatica)
3. costruzione dell'albero sintattico (interpretazione)
4. esecuzione eseguendo le funzioni utente

L'esecuzione avviene eseguendo delle funzioni definite dall'utente, in momenti separati dall'interpretazione vera e propria. In questo modo si ottengono buone prestazioni di velocità di esecuzione in quanto il programma viene prima interpretato e poi eseguito. L'overhead per ogni funzione è molto piccola.

E' comunque possibile separare la fase di parsing dalla fase di esecuzione.

Come si usa

Vediamo prima alcuni concetti base e poi alcuni semplici esempi di utilizzo. A questo proposito si esamini il codice dell'esempio fornito.

I membri pubblici della classe Hlang_cls sono:

- `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> ¶ms,`

```
vector<int> &paramstype)
```

Innanzitutto occorre derivare la classe affinché sia abbia accesso ai membri virtuali soprattutto per quanto riguarda il membro per la gestione della semantica.

La funzione più importante è quella che si incarica di gestire la semantica, ossia il membro `ExeFuncDispatch` che deve implementare tutte le funzioni utente che danno un significato alle varie parti del programma.

Utilizzo della classe

Quando viene stanziata la classe, il costruttore riceve un nome di file, ossia il file che contiene la descrizione della grammatica. Ad esempio il costruttore della classe `example_cls` sarà:

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

Quando verrà stanziata la classe `example_cls` il costruttore `Hlang_cls` caricherà la grammatica descritta nel file `grammar.g`.

Membro per le funzioni semantiche

Occorre poi implementare la funzione `ExeFuncDispatch` che verrà richiamata una volta interpretato il programma utente. Questa funzione ha i seguenti parametri:

<code>pos</code>	indica con un numero la posizione logica dell'istruzione interpretata
<code>func</code>	ogni funzione associata alla semantica di un dato simbolo è rappresentata da un numero intero positivo (vedere anche la grammatica. Vi sono comunque due funzioni che vengono sempre eseguite: - 1: prima di iniziare l'esecuzione - 2: al termine del programma
<code>params</code>	parametri passati alla funzione. Sono passati come vettore di stringhe. E' compito delle funzioni utente convertire nel dato opportuno.
<code>paramstype</code>	tipo del parametro passato come specificato dalla grammatica. E' passato come vettore di interi.

Un modo semplice per implementare le funzioni semantiche è il seguente

```
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);
}

```

Questo membro deve restituire un intero che avrà il seguente significato a seconda del suo valore:

Se il valore è maggiore o uguale a 0 indica quale sarà la prossima istruzione da eseguire, utile nel caso si debbano implementare istruzioni come `for`, `while` o `goto` che necessitano di poter saltare in un punto particolare del programma.

Se invece il valore ritornato è minore di 0, acquisisce un significato particolare

`HLANG_RET_NEXT`

esegue la prossima istruzione come risulta da programma

`HLANG_RET_EXIT`

termina l'esecuzione

`HLANG_RET_DEC_LEVEL`

decrementa il livello di nidificazione del blocco (vedere più avanti)

`HLANG_RET_INC_LEVEL`

incrementa il livello di nidificazione del blocco (vedere più avanti)

`HLANG_RET_ERROR`

esce indicando un errore

`HLANG_RET_NEXTPASS`

in caso di implementazione di linguaggi che richiedono più passate, indica quando passare al successivo. Il passo corrente è indicato nella variabile `pass` e parte da 1 per la prima passata.

NOTA: Ad ogni passata il programma viene rieseguito daccapo!

`HLANG_RET_NEXTIGNORELEVEL`

manda in esecuzione la prossima istruzione senza tenere conto del livello di nidificazione del blocco

Se si desiderano gestire le passate si può ad esempio scrivere così:

```

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);
}

```

In questa funzione è possibile intervenire sul funzionamento del sistema per eseguire il programma. Esistono infatti delle proprietà che possono essere lette/modificate:

pass

specifica la passata corrente.

next_pass

posta a true, specifica che si desidera effettuare un'altra passata del programma. Esiste anche come valore di ritorno, ma questo consente di alterare il funzionamento potendo ritornare un'altro valore se necessario.

ignorelevel_lock

consente di sovrascrivere il livello ingorando quello che ritorna questa funzione. Se ad esempio si stanno cercando delle label dentro il programma per i salti con una funzione tipo goto, è utile ignorare i blocchi, per cui il livello non deve essere alterato, in quanto le label sono assolute e non relative al blocco in cui si trovano.

stop_script

ferma l'esecuzione del programma se posto a true

Uno sguardo allo scanner

La classe implementa uno scanner che permette senza sforzo di recuperare i token dal programma. Tale membro è virtuale per cui può essere implementato secondo le proprie esigenze, ma per una buona parte delle applicazioni dovrebbe essere sufficiente quello qui fornito.

Lo scanner preleva i token specificati nella grammatica, sia i terminali che i macroterminali. Scorre il programma alla ricerca della produzione che genera il token più lungo. Il suo funzionamento è simile a `flex`, ovviamente con le limitazioni del caso.

Descrizione della grammatica

La grammatica del linguaggio da implementare deve venire descritta in un file di testo. Di seguito un esempio:

```
&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 %Talphnum   ^\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 )

```

vediamo nel dettaglio il significato di tutto ciò.

Al momento non soffermiamoci su che tipo di grammatica si voglia descrivere.

Sono permessi commenti, ma solo di intere linee e iniziano per # .

All'inizio del file sono specificate alcune definizioni interne mediante delle variabili che sono utili per semplificare il lavoro dell'utente. Le variabili interne iniziano per & e sono le seguenti:

&comment_line *str*

permette di specificare con una stringa quale simbolo viene usato nel programma per definire commenti di intere linee, come le `\\` del C++ ad esempio. Queste righe verranno ignorate in fase di caricamento del programma.

&comment_block_start *str*

permette di definire il simbolo di inizio per i blocchi di commento, come il `/*` del C. Il contenuto tra questi blocchi sarà ignorato in fase di caricamento del programma.

&comment_block_end *str*

permette di definire il simbolo di fine per i blocchi di commento, come il

*/ del C. Il contenuto tra questi blocchi sarà ignorato in fase di caricamento del programma.

`&terminal ntype %Tname regex`

Specifica un simbolo terminale. In realtà non sono proprio simboli terminali, in quanto la loro forma è specificata dall'espressione regolare. Questi simboli sono nel seguito chiamati `macroterminali`. Devono essere specificati tre parametri:

<code>ntype</code>	numero intero che indica il tipo di macroterminale. E' un numero tra 1 e 999 ed è il valore che verrà passato alla funzione semantica nel parametro chiamato <code>paramstype</code>
<code>%Tname</code>	nome del simbolo che verrà utilizzato dalla grammatica. Deve iniziare con <code>%T</code>
<code>regex</code>	espressione regolare. Si utilizza la sintassi standard utilizzata dalla libreria <code>regex</code> . Per maggiori dettagli riferirsi al manuale della libreria <code>regex.h</code>

`&delimiters [str]`

Permette di specificare quali separatori per i token vengono utilizzati nel programma. Se non viene specificato il default è la stringa `\t\n\r`. Si noti che le parentesi `[e]` sono necessarie

I simboli terminali utilizzati, vengono prelevati direttamente dalla descrizione delle produzioni grammaticali, per cui non si devono specificare, ma solo utilizzare.

I simboli non terminali devono iniziare per `%` quando vengono utilizzati nelle produzioni, a cui non deve seguire una `T`, che verrebbero altrimenti confusi con i simboli macroterminali. Tale simbolo non deve essere messo invece nelle definizioni del simbolo non terminale.

Le produzioni avranno questo formato:

`non_terminal_symb := production semantic`

Se vi sono delle produzioni in `OR` occorre ripetere il simbolo non terminale e descrivere la produzione alternativa come nell'esempio il simbolo non terminale `statement`.

per quanto riguarda la semantica, la più semplice è la riga vuota!

Ciascuna produzione, può avere solo una funzione semantica associata, specificata dal simbolo `$Fn`, dove `n` è un numero intero positivo. Ad esempio, osserviamo la seguente riga:

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

quando il parser individua questa produzione, la funzione preposta per l'esecuzione chiamerà il membro `ExecFuncDispatch` per far eseguire la funzione 15 senza alcun parametro, fatto specificato dalle `()` senza spazi tra le due parentesi.

Analizziamo ora la riga seguente

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

In questo caso verrà richiamata la funzione 10, ma stavolta verranno passati tre parametri. Il numero all'interno delle parentesi indica la posizione del simbolo nella grammatica. Il numero 1 indica il primo simbolo nella produzione, il 2 il secondo e così via.

Si possono quindi verificare 2 casi:

1. il simbolo riferito è un terminale o macroterminale: il parametro della funzione conterrà proprio il simbolo.
2. il simbolo è un non terminale: il parametro allora è la posizione del simbolo nell'albero sintattico

nell'esempio sopra, supponiamo che corrisponda ad una riga di programma di questo tipo, che ovviamente corrisponde a questa produzione:

```
alpha == 23
```

abbiamo i seguenti parametri: il primo (posizione 1) sarà `alpha`, il secondo (posizione 2) sarà `==` e il terzo (posizione 3) sarà `23`.

Vediamo un altro esempio:

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

Come si nota, la funzione 4 riceve un parametro che corrisponde al simbolo non terminale `%block`. verrà quindi passato nella lista dei parametri la posizione della primo simbolo terminale individuato nella produzione che descrive il simbolo non terminale `%block`.

Infatti il parser, via via che analizza il programma, costruisce in sequenza un archivio di tutti i simboli terminali trovati, in modo da costruire l'albero sintattico. Quindi il valore passato alla funzione semantica quando il simbolo è un non terminale, si può vedere come un puntatore all'albero sintattico in quel punto.

La produzione sopra descrive un costrutto del tipo

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

per cui, in riferimento a quanto detto, il parametro corrispondente alla posizione 6 della grammatica in fase di esecuzione sarà la posizione della prima istruzione che costituisce `%block`.

Ci sono delle particolarità nella semantica. supponiamo di voler implementare un linguaggio che abbia possibilità di nidificare dei costrutti. Per fare ciò utilizziamo una variabile che indica il livello di nidificazione in modo che la funzione che esegue il programma sappia quale sia la prossima istruzione da eseguire corrispondente al livello giusto. Facciamo un esempio:

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

questa produzione dice che quando si incontra il simbolo `{` si deve incrementare il livello di nidificazione (`$+`) ed eseguire la funzione 100 che non ha parametri.

La corrispondente riga che indica come si esce da un blocco nidificato è ad esempio

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

in questo caso dice che quando si incontra il simbolo `}` si deve scendere di un livello di nidificazione (`$*-`) e richiamare la funzione 101 che non ha parametri.

Si noti che il livello può essere incrementato/decrementato prima o dopo essere entrati o usciti dal blocco.

Con i simboli `$+` e `$-` il livello di indentazione viene incrementato/decrementato prima di entrare/usciti nel blocco, mentre coi simboli `$*+` e `$*-` viene incrementato/decrementato dopo che si è entrati/usciti.

Nel caso dell'esempio, il simbolo `{` farà parte del livello successivo ed anche il simbolo `}` perché abbiamo indicato di decrementare il livello dopo la memorizzazione con `$*-`.