P O L I T E C N I C O D I B A R I

I FACOLTA' DI INGEGNERIA

CORSO DI LAUREA IN INGEGNERIA INFORMATICA SPECIALISTICA

Tema d'anno per l'esame di

Compilatori ed Interpreti

Compilatore per il linguaggio

Docente: Prof. Giacomo Piscitelli

Autori: Matteo Marco RINALDI

Giuseppe VILLANI

A.A. 2010/2011

INDICE

INTRODUZIONE

In questo lavoro abbiamo realizzato un parser ed un scanner PHP, analizzandone nello specifico il funzionamento basandoci sulle specifiche della versione PHP 5.2.16, rilasciato il 16 Dicembre 2010 dalla community ufficiale che sviluppa questo linguaggio di scripting.

La scelta del linguaggio PHP è stata veicolata dall'uso "quotidiano" di tale linguaggio web-oriented nei nostri studi, nei nostri progetti e nei nostri lavori; l'implementazione del compilatore ci ha aiutato a capire in maniera molto approfondita le problematiche di gestione e le peculiarità del linguaggio stesso a basso livello: ci ha supportato, anche tramite il corso, nel comprendere i sorgenti, le funzioni e le librerie necessarie per la creazione del compilatore vero e proprio, insegnandoci anche ad usarlo nella maniera più corretta a tutti i livelli.

Siamo partiti studiando e interpretando i sorgenti ufficiali del linguaggio, pieni di chiamate a funzione e librerie esterne, che hanno il compito di permettere l'interpretazione e l'esecuzione del linguaggio su ogni macchina: abbiamo ritenuto non essere utili questi sorgenti nella realizzazione del nostro progetto se non come punto di partenza prettamente "teorico".

Abbiamo creato da zero la grammatica (parser) e la sintassi (scanner) generando regole per la maggior parte dei costrutti (evitando classi ed array per questioni di eccessiva complessità), ed espressioni regolari per tutte le tipologie di token accettate dal linguaggio.

Per quanto riguarda la grammatica, abbiamo generato dapprima la BNF del linguaggio tramite tool software Gold Parser Builder, traducendola poi in linguaggio accettato dal tool software di generazione automatica di parser YACC, gestendo molti casi d'errore. Servendoci successivamente del manuale online PHP per le keywords e i token, abbiamo generato l'analizzatore lessicale servendoci del tool FLEX. Abbiamo inoltre implementato anche un analizzatore semantico che coopera con la symbol table generata dal parser, per la gestione degli errori non esprimibili in produzioni grammaticali.

L'ambiente di lavoro utilizzato è stato Linux, nello specifico la distribuzione Ubuntu 10.04.

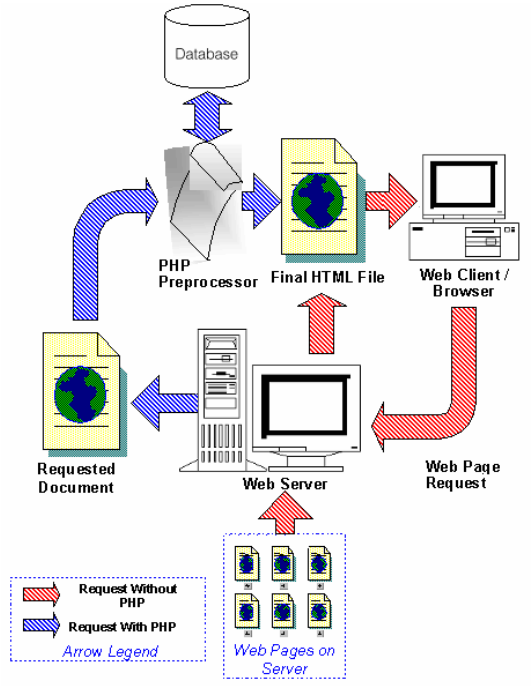
La versione di Flex usata è la 2.5.35, la versione di Yacc/Bison utilizzata è la 2.4.1.

IL LINGUAGGIO PHP

PHP (acronimo ricorsivo di "PHP: Hypertext Preprocessor", preprocessore di ipertesti) è un linguaggio di scripting interpretato, con licenza open source e libera (ma incompatibile con la GPL), originariamente concepito per la programmazione Web ovvero la realizzazione di pagine web dinamiche.

Attualmente è utilizzato principalmente per sviluppare applicazioni web lato server ma può essere usato anche per scrivere script a riga di comando o applicazioni standalone con interfaccia grafica, e consente di interagire praticamente con qualsiasi tipo di database (MySql, PostgreSql, Sql Server, Oracle, SyBase, Access e altri).

L'elaborazione di codice PHP sul server produce codice HTML da inviare al browser dell'utente che ne fa richiesta. Il vantaggio dell'uso di PHP e degli altri linguaggi Web come ASP e .NET rispetto al classico HTML derivano dalle differenze profonde che sussistono tra Web dinamico e Web statico.



Nato nel 1994 ad opera del danese Rasmus Lerdorf, PHP era in origine una raccolta di script CGI che permettevano una facile gestione delle pagine personali. Il significato originario dell'acronimo era Personal Home Page (secondo l'annuncio originale di PHP 1.0 da parte dell'autore sul newsgroup comp.infosystems.www.authoring.cgi).

Il pacchetto originario venne in seguito esteso e riscritto dallo stesso Lerdorf in C, aggiungendo funzionalità quali il supporto al database mSQL e prese a chiamarsi PHP/FI, dove FI sta per Form Interpreter (interprete di form), prevedendo la possibilità di integrare il codice PHP nel codice HTML in modo da semplificare la realizzazione di pagine dinamiche. In quel periodo, 50.000 domini Internet annunciavano di aver installato PHP.

A questo punto il linguaggio cominciò a godere di una certa popolarità tra i progetti open source del web, e venne così notato da due giovani programmatori: Zeev Suraski e Andi Gutmans. I due collaborarono nel 1998 con Lerdorf allo sviluppo della terza versione di PHP (il cui acronimo assunse il significato attuale) riscrivendone il motore che fu battezzato Zend da una contrazione dei loro nomi. Le caratteristiche chiave della versione PHP 3.0 frutto del loro lavoro, erano la straordinaria estensibilità, la connettività ai database e il supporto iniziale per il paradigma a oggetti. Verso la fine del 1998 PHP 3.0 era installato su circa il 10% dei server web presenti su Internet.

PHP diventò a questo punto talmente maturo da competere con ASP, linguaggio lato server analogo a PHP sviluppato da Microsoft, e cominciò ad essere usato su larga scala. La versione 4 di PHP venne rilasciata nel 2000 e prevedeva notevoli migliorie. Attualmente siamo alla quinta versione, sviluppata da un team di programmatori, che comprende ancora Lerdorf, oltre a Suraski e Gutmans.

La popolarità del linguaggio PHP è in costante crescita grazie alla sua flessibilità: nel Giugno 2001, ha superato il milione di siti che lo utilizzano. Nell'ottobre 2002, più del 45% dei server Apache usavano PHP.

Nel gennaio 2005 è stato insignito del titolo di "Programming Language of 2004" dal TIOBE Programming Community Index, classifica che valuta la popolarità dei linguaggi di programmazione sulla base di informazioni raccolte dai motori di ricerca.

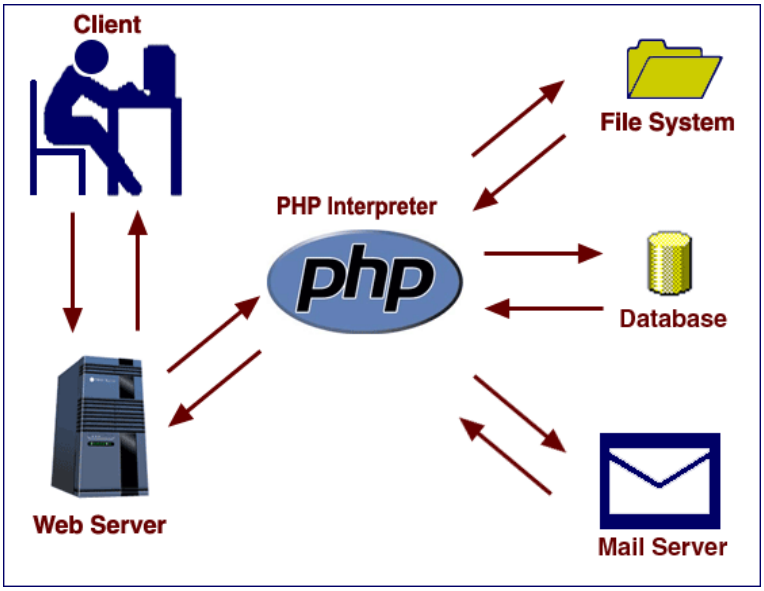
Nel 2005 la configurazione LAMP (Linux, Apache, MySQL, PHP) supera il 50% del totale dei server sulla rete mondiale.

PHP riprende per molti versi la sintassi del C, come peraltro fanno molti linguaggi moderni, e del Perl. È un linguaggio a tipizzazione debole e dalla versione 5 migliora il supporto al paradigma di programmazione ad oggetti. Certi costrutti derivati dal C, come gli operatori fra bit e la gestione di stringhe come array, permettono in alcuni casi di agire a basso livello; tuttavia è fondamentalmente un linguaggio di alto livello, caratteristica questa rafforzata dall'esistenza delle sue moltissime API, oltre 3.000 funzioni del nucleo base. PHP è in grado di interfacciarsi a innumerevoli database tra cui MySQL, PostgreSQL, Oracle, Firebird, IBM DB2, Microsoft SQL Server, solo per citarne alcuni, e supporta numerose tecnologie, come

XML, SOAP, IMAP, FTP, CORBA. Si integra anche con altri linguaggi/piattaforme quali Java e .NET e si può dire che esista un wrapper per ogni libreria esistente, come CURL, GD, Gettext, GMP, Ming, OpenSSL ed altro.

Fornisce un'API specifica per interagire con Apache, nonostante funzioni naturalmente con numerosi altri server web. È anche ottimamente integrato con il database MySQL, per il quale possiede più di una API. Per questo motivo esiste un'enorme quantità di script e librerie in PHP, disponibili liberamente su Internet. La versione 5, comunque, integra al suo interno un piccolo database embedded, SQLite.

Dispone di un archivio chiamato PEAR che mette a disposizione un framework di librerie riusabili per lo sviluppo di applicazioni PHP e di PECL che raccoglie tutte le estensioni conosciute scritte in C.



PHP è un linguaggio la cui funzione fondamentale è quella di produrre codice HTML. Siccome però PHP è un linguaggio di programmazione, abbiamo la possibilità di analizzare diverse situazioni (l'input degli utenti, i dati contenuti in un database) e di decidere, di conseguenza, di produrre codice HTML condizionato dai risultati dell'elaborazione. Per questo con PHP è possibile creare pagine dinamiche.

La prima cosa da sapere è come fa PHP (inteso come interprete) a riconoscere il codice php contenuto nel file che sta analizzando. Il codice php infatti deve essere compreso fra appositi tag di apertura e di chiusura, che sono i seguenti:

<?php → tag di apertura

?> → tag di chiusura

Tutto ciò che è contenuto fra questi tag deve corrispondere alle regole sintattiche del PHP, ed è codice che sarà eseguito dall'interprete e non sarà inviato direttamente al browser.

Le **variabili** sono componenti fondamentali di qualsiasi linguaggio di programmazione, in quanto ci consentono di trattare i dati del nostro programma senza sapere a priori quale sarà il loro valore.

In PHP possiamo scegliere il nome delle variabili usando lettere, numeri o underscore, il primo carattere del nome, però, non deve essere un numero.

PHP è sensibile all'uso delle maiuscole e delle minuscole (case sensitive): di conseguenza, se scriviamo due volte un nome di variabile usando le maiuscole in maniera differente, si tratterà di due variabili distinte.

Il nome delle variabili è preceduto dal simbolo dollaro ($) e ha una caratteristica che rende il PHP molto più flessibile rispetto ad altri linguaggi di programmazione: non richiede, infatti, che le variabili vengano dichiarate prima del loro uso. Possiamo quindi permetterci di riferirci ad una variabile direttamente con la sua valorizzazione.

Una variabile può contenere diversi tipi di valori, ognuno dei quali ha un comportamento ed un'utilità differente e che PHP, a differenza di altri linguaggi, associa al valore e non alla variabile ed effettua conversioni automatiche dei valori nel momento in cui siano richiesti tipi di dato differenti.

- ***Valore booleano***

I tipi di dato boolean servono per indicare i valori vero o falso all'interno di espressioni logiche. Il tipo booleano è associato alle variabili che contengono il risultato di un'espressione booleana oppure i valori true e false.

- ***Intero***

Un numero intero, positivo o negativo, il cui valore massimo (assoluto) può variare in base al sistema operativo.

• ***Virgola mobile***

Un numero decimale (a volte citato come "double" o "real"). Anche in questo caso la dimensione massima dipende dalla piattaforma. Normalmente comunque si considera un massimo di circa 1.8e308 con una precisione di 14 cifre decimali. Si possono utilizzare le seguenti sintassi:

```
<?php
        $vm1 = 4.153;          → 4,153
        $vm2 = 3.2e5;          → 3,2 * 10^5, cioè 320.000
        $vm3 = 4E-8;           → 4 * 10^-8, cioè 4/100.000.000 = 0,00000004
?>
```

• ***Stringa***

Una stringa è un qualsiasi insieme di caratteri, senza limitazione, contenuto all'interno di una coppia di apici singoli o doppi (se all'interno della stringa è presente un nome di variabile).

Può capitare che una stringa debba contenere a sua volta un apice o un paio di virgolette, in questo caso abbiamo bisogno di un sistema per indicare che quel carattere fa parte della stringa e non è il suo delimitatore. In questo caso si usa il 'carattere di escape', (backslash: \), che viene usato anche come 'escape di sè stesso', nei casi in cui vogliamo esplicitamente includerlo nella stringa.

• ***Array***

Possiamo considerare un array come una variabile complessa, che contiene una serie di valori, ciascuno dei quali caratterizzato da una chiave, o indice che lo identifica univocamente. Per recuperare un determinato valore dalla variabile che contiene l'array, è sufficiente specificare il suo indice all'interno di parentesi quadre dietro al nome della variabile.

Gli **operatori** sono un altro degli elementi di base di qualsiasi linguaggio di programmazione, in quanto ci consentono non solo di effettuare le tradizionali operazioni aritmetiche, ma più in generale di manipolare il contenuto delle nostre variabili.

• ***Operatore di assegnazione***

```
$nome = 'Giorgio';
```

Il simbolo '=' serve ad assegnare alla variabile $nome il valore 'Giorgio'. In generale, prendiamo ciò che sta alla destra del segno di ugualianza ed assegnamo lo stesso valore a ciò che sta a sinistra.

• ***Operatori aritmetici***

```
+ -     → addizione/sottrazione
*       → moltiplicazione
/       → divisione
%       → modulo
```

• ***Concatenare le stringhe***

```
$nome = 'pippo';
$stringa1 = 'ciao ' . $nome;
```

• ***Incremento e decremento***

```
$a++; ++$a;     → incrementa di 1
$a--; --$a;     → decrementa di 1
```

• ***Operazioni di confronto***

```
==      → uguale
!=      → diverso
===     → identico (cioè uguale e dello stesso tipo)
>       → maggiore
>=      → maggiore o uguale
<       → minore
<=      → minore o uguale
```

• ***Operatori logici***

Or &&	→ valuta se almeno uno dei due operatori è vero;
And ||	→ valuta se entrambi gli operatori sono veri;
Xor	→ viene chiamato anche 'or esclusivo', valuta se uno solo dei due è vero: l'altro deve essere falso;
!	→ è l'operatore 'not' e vale come negazione. Si usa con un solo operatore: è vero quando l'operatore è falso.

Una **funzione** è un insieme di istruzioni che hanno lo scopo di eseguire determinate operazioni. Le funzioni possono essere incorporate nel linguaggio oppure essere definite dall'utente.

La sintassi fondamentale con la quale si richiama una funzione è molto semplice. Si tratta semplicemente di indicare il nome della funzione, seguito da parentesi tonde, che devono contenere i parametri da passare alla funzione, obbligatoriamente indicate anche se non ci sono parametri. Nel caso poi in cui la funzione restituisca un valore, possiamo indicare la variabile in cui immagazzinarlo:

```
$valore = nome_funzione();
```

Alcune fra le funzioni utilizzate nel nostro progetto PHP:

• empty(valore): verifica se la variabile che le passiamo è vuota;

• isset(valore): verifica se la variabile è definita;

• eval(codice): permette di inserire stralci di codice PHP valido.

In PHP abbiamo la possibilità di definire delle funzioni che ci permettono di svolgere determinati compiti in diverse parti del nostro script, semplicemente richiamando la porzione di codice relativa, alla quale avremo attribuito un nome che identifichi la funzione stessa.

La definizione della funzione avviene attraverso la parola chiave function, seguita dal nome, e dalle parentesi che contengono i parametri che devono essere passati alla funzione. Di seguito, contenuto fra parentesi graffe, ci sarà il codice che viene eseguito ogni volta che la funzione viene richiamata. Il nome della funzione deve essere necessariamente univoco.

ANALISI LESSICALE

2.1 Flex

I linguaggi umani europei sono codificati utilizzando un ristretto insieme di simboli, i caratteri dell'alfabeto; tuttavia sono troppo pochi per descrivere un insieme di concetti sufficiente per l'uso: non si considerano infatti i singoli caratteri ma i loro raggruppamenti in parole.

Analogamente, nei linguaggi formali, i programmi sono codificati utilizzando un ristretto insieme di simboli, di solito l'ASCII, e anche in questo caso si considerano unità non tanto i singoli caratteri ma dei raggruppamenti significativi, i token appunto. Un compilatore tuttavia riceve in input una sequenza di caratteri ASCII ed il suo primo compito è quello di "spezzettare" l'input in token.

L'operazione di tokenizzazione può essere svolta in vari modi. Si può scrivere direttamente un programma in C, o si può utilizzare il formalismo delle grammatiche anche per riconoscere i token. Tuttavia il problema della tokenizzazione è sufficientemente noto da poter essere automatizzato e abbastanza complesso da essere scomodo da implementare a mano. Il meccanismo delle grammatiche è molto potente, ma applicarlo alla tokenizzazione si rivela inefficiente.

Tradizionalmente si divide l'analisi sintattica in due fasi, la prima detta analisi lessicale (che corrisponde alla tokenizzazione dell'input) e la seconda di analisi sintattica vera e propria. Per la generazione di analizzatori lessicale si utilizza spesso il tool Lex/Flex, che è progettato specificamente per essere utilizzato insieme allo Yacc. Infatti molte assunzioni del codice generato da Lex si sposano bene con quelle dello Yacc. Per esempio l'analizzatore lessicale prodotto da Flex è una funzione C chiamata yylex(), che è esattamente quella che si aspetta Yacc per l'analizzatore lessicale.

Flex (fast lexical analyzer generator) è un free software scritto originariamente in C, nel 1987, da Vern Paxson.

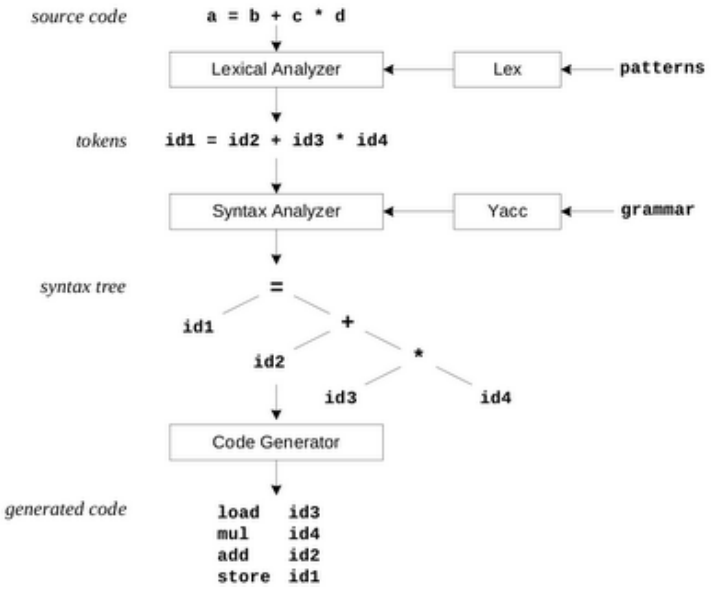
Flex utilizza per la specifica dell'analizzatore lessicale le espressioni regolari, che sono un formalismo più efficiente delle grammatiche ma meno potente. La distinzione tra grammatiche e espressioni regolari sta nel fatto che le espressioni regolari non sono in grado di riconoscere strutture sintattiche ricorsive, mentre questo non è un problema per le grammatiche. Una struttura sintattica come le parentesi bilanciate, che richiede che le

parentesi aperte siano nello stesso numero di quelle chiuse non può essere riconosciuta da un analizzatore lessicale, e per questo scopo si ricorre all'analizzatore sintattico. Invece costanti numeriche, identificatori o keyword vengono normalmente riconosciute dall'analizzatore lessicale.


```
C and scanner declarations
%%
Token definitions and actions
%%
C subroutines
```

L'input di Lex viene diviso in tre parti, separate da %%. La prima parte comprende delle definizioni, che sono delle sottoespressioni regolari, da utilizzare più avanti, oltre alla definizione di token multi carattere tramite inclusione del file y.tab.h generato dal parser. In generale le espressioni regolari possono essere anche molto complesse, quindi queste definizioni sono indispensabili per semplificare le espressioni regolari vere e proprie. La seconda parte comprende le espressioni regolari che vengono riconosciute dall'analizzatore lessicale; si può notare che alcune utilizzano definizioni dalla prima parte. Associata ad ogni espressione regolare c'è un pezzo di codice C che viene eseguito ogni volta che una espressione regolare viene riconosciuta. La terza parte comprende solo codice di supporto in C. Parti di codice possono essere inserite sia nella prima parte (inserendolo tra %{ e %}) che nella seconda, inserendolo tra { } immediatamente dopo ogni espressione regolare che si vuole riconoscere.

Lex genera una funzione, yylex(), che legge lo standard input e viene ripetutamente invocata dal parser ogni volta che è richiesto un nuovo token.



Una volta che un token viene riconosciuto, abbiamo varie possibilità: possiamo ignorarlo (come facciamo per gli spazi bianchi) e passare al token successivo; oppure possiamo ritornare il codice del token riconosciuto. Quando viene ritornato un token, la funzione yylex() termina ma sarà richiamata dal parser quando avrà bisogno di un altro token.

Notiamo che può essere necessario fare qualche azione supplementare oltre a ritornare un token o ignorarlo. Se ad esempio incontrassimo un newline, potremmo incrementare il contatore delle linee in input. Molto più importante è il fatto che di certi token occorre conoscere qualcosa di più oltre al loro tipo. Per esempio, di una variabile non basta sapere che è una variabile: occorre sapere quale è. Lex mantiene in un buffer il testo letto dall'input e riconosciuto, buffer accessibile all'utente tramite le variabili char* yytext e int yyleng.

Ad ogni token, come pure ad ogni simbolo grammaticale, è possibile associare un valore. I valori associabili ad un token sono definiti da una union, dichiarata nel sorgente Yacc. Nel nostro caso dichiareremo anche un campo str di tipo char*. La variabile yylval, che è una union, è utilizzata per passare al parser il valore associato al token corrente; per variabili, costanti e stringhe provvederemo a passare informazioni supplementari, usando i campi di yylval.

Il testo riconosciuto da una espressione regolare si trova momentaneamente nel buffer yytext, ma dobbiamo provvedere a copiarlo da qualche parte per passarlo al parser; questa operazione è effettuata di solito assegnando valori coerenti alla classe yylval. Nella terza sezione può anche trovare posto la funzione yywrap(), che viene invocata da yylex() quando incontra la fine dell'input per decidere il da farsi. Se ritorna 1, vuol dire che l'input è proprio finito. Altrimenti la yywrap() può prendere provvedimenti, come aprire un altro file e collegarlo allo standard input, ritornando 0 per dire a yylex() di continuare a leggere.

Il testo non riconosciuto tramite nessuna espressione regolare viene ricopiato in uscita, carattere per carattere.

2.2 File Php-scanner.l

```
%x IN_PHP

%x DOUBLE_QUOTES

%x SINGLE_QUOTE

%x BACKQUOTE

%x COMMENT
```

In questa sezione sono elencate le START CONDITION utilizzate nell'analisi lessicale. Una start condition serve per limitare lo scope di determinate regole, o per modificare la modalità di gestione del file di input

```
<INITIAL>"<?php"[ \n\r\t] {
        if (yytext[yyleng-1] == '\n') {
                lineno++;
        }
        BEGIN(IN_PHP);
}

<IN_PHP>("?>"|"</script"{WHITESPACE}*">")([\n]|"\r\n")? {
        BEGIN(INITIAL);
}
```

La start condition INITIAL è quella di partenza, corrisponde allo stato zero dell'automa associato allo scanner: una regola sarà matchata solo se la start condition che la precede è attiva. Una volta che la prima regola dell'esempio viene matchata, la direttiva BEGIN(IN_PHP) porta lo stato dell'automa nello stato IN_PHP rendendo valide tutte le regole precedute dalla stessa start condition. "Lineno" è la variabile che conta le righe di codice esaminate.

```
LNUM [0-9]+

DNUM ([0-9]*[\.][0-9]+)|([0-9]+[\.][0-9]*)

HEXNUM "0x"[0-9a-fA-F]+

EXPONENT_DNUM (({LNUM}|{DNUM})[eE][+-]?{LNUM})
```

```
LABEL   [a-zA-Z_\x7f-\xff][a-zA-Z0-9\x7f-\xff]*

T_STRING {LABEL}

T_VARIABLE "$"{LABEL}
```

Qui mostriamo alcuni alias presenti nella prima parte del file Flex, utili per la semplificazione delle definizioni delle espressioni regolari presenti nella seconda sezione.

```
<IN_PHP>{LABEL} {
        fprintf(tk,"T_STRING\n");
        yylval.id = (char *)strdup(yytext);
        return T_STRING;
}

<IN_PHP>{LNUM}|{HEXNUM} {
        fprintf(tk,"T_LNUMBER\n");
        yylval.numero = atoi(yytext);
        return T_LNUMBER;
}

<IN_PHP>{DNUM}|{EXPONENT_DNUM} {
        fprintf(tk,"T_DNUMBER\n");
        yylval.numero = atof(yytext);
        return T_DNUMBER;
}
```

Esempio di espressioni regolari nella sezione due, dove si nota l'utilizzo di alias definiti nella prima parte, le start condition e il ritorno della tipologia di token trovata. In questo stralcio di codice, ogni volta che viene matchata l'espressione regolare viene ritornato al parser il tipo di token corrispondente, viene memorizzata la tipologia di token all'interno di un file binario tramite il puntatore allo stream di input "tk" (che servirà successivamente come controllo) e viene settata la variabile yylval. Questa variabile viene definita tramite una "union" nel codice del parser, per il nostro caso di studio abbiamo previsto che abbia campi "numero" e "id".

```
%%

<IN_PHP>{T_VARIABLE_ERR} {
        fprintf(tk,"T_VARIABLE\n");
        yylval.id = (char *)strdup(yytext+2);
        yyerror("\033[01;33mWARNING: e' stato corretto un errore lessicale\033[00m\n");
        count--;
        warn++;
        return T_VARIABLE;
}

<IN_PHP>{T_VARIABLE_ERR2} {
        yyerror("\033[01;33mERRORE LESSICALE: variabile non puo´ cominciare con
numeri\033[00m\n");
        return T_VARIABLE;

}
```

In questo pezzo di codice viene mostrato come vengano riconosciuti gli errori lessicali: nella prima espressione regolare "t_variable_err" vengono riconosciuti i nomi di variabili che iniziano per un numero (non consentito dal php), questa imprecisione viene corretta considerando la restante parte della stringa (senza numero) e inserita nella variabile yylval, viene ritornato comunque il token "T_VARIABLE", viene decrementato il numero degli errori (perché è stato corretto) ma viene incrementato il numero di warning. La seconda espressione regolare invece riconosce nomi di variabile che iniziano per una o più cifre, restituendo comunque il token "T_VARIABLE" per continuare il parsing (come prevede la modalità "error production"), incrementando il numero di errori.

```
%%

segnalazione(){
if(warn>0)printf("          \033[01;33me %d warning\033[00m\n\n",warn);
else printf("\n");
}
```

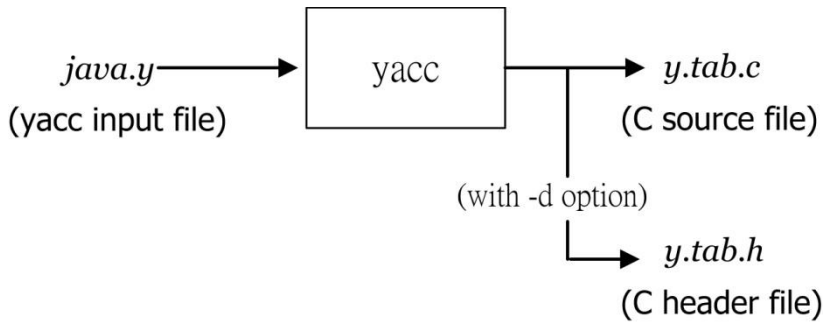
Nella terza sezione del file viene definita la funzione "segnalazione()", che verrà riportata nel sorgente finale e servirà per visualizzare gli eventuali warning in fase di compilazione.

ANALISI SINTATTICA

3.1 YACC/Bison

Il tool Yacc (acronimo di Yet Another Compiler-Compiler) è uno strumento che a partire da una specifica grammaticale context free di un linguaggio scritta in una appropriata sintassi genera automaticamente un parser LALR del linguaggio stesso. In particolare Yacc è capace di generare una traduzione input/output che oltre a fare l'analisi sintattica di costrutti input ne fa anche la traslazione in un certo output come definito dalle azioni.

L'architettura software relativa al contesto d'uso di Yacc è la seguente:



L'utente deve concentrarsi esclusivamente sulla definizione della specifica sorgente Yacc (file .y in cui è scritta la grammatica) , tutto il resto è infatti implementato automaticamente. Il formato generale di un programma sorgente Yacc è costituito da tre sezioni e ha la seguente struttura:

```
DICHIARAZIONI

%%

REGOLE DI TRASLAZIONE

%%

ROUTINES AUSILIARIE
```

Il cuore della specfica Yacc è rappresentato dalla sezione "REGOLE DI TRASLAZIONE" in quanto in essa è contenuta la grammatica vera e propria.

- Sezione **"DICHIARAZIONI"**

Tale sezione è costituita di due parti. La prima contiene tutto il blocco dichiarazioni in C. Sostanzialmente vi vengono inserite le normali dichiarazioni di variabili che saranno usate nelle sezioni delle Regole e dei Programmi, e in più vi possono essere direttive di compilazione del tipo #define o #include. Questa parte è delimitata dai simboli %{ e %}. Nella seconda parte invece si dichiarano i token della grammatica atraverso la direttiva Yacc %token seguita dalla lista dei token.

- Sezione **"REGOLE DI TRASLAZIONE"**

Si rappresentano seguendo la seguente sintassi:

la produzione: <ParteSinistra> ==> <ParteDestra>

in formato Yacc è scritta nel seguente modo:

<ParteSinistra> : <ParteDestra>;

All'interno di una produzione, Yacc considera come terminali o stringhe di caratteri dichiarate come token nella sezione Dichiarazioni oppure come singoli caratteri racchiusi tra apici. Questi ultimi tipi di terminali il parser se li aspetta così come sono nell'input, invece i valori dei terminali dichiarati come token saranno passati al parser dall'anlizzatore lessicale.

Ogni stringa di caratteri che appare nelle produzioni e che non è stata dichiarata come token viene assunta da Yacc essere non terminale.

Il formato generale di una produzione Yacc è il seguente:

<ParteSinistra> : <ParteDestra> {AzioneSemantica};

Dove per azione semantica si intende una sequenza di istruzioni scritte in linguaggio C e racchiuse tra parentesi graffe.

Nelle produzioni Yacc i simboli grammaticali hanno attributi semantici, cioè valori, e le azioni servono proprio a gestire questi valori. Yacc usa la pseudo-variabile $$per far riferimento al valore dell'attributo semantico del non-terminale sinistro di una produzione, e le pseudo-variabili $i per il valore dell'attributo dell'i-esimo simbolo grammaticale della parte destra.

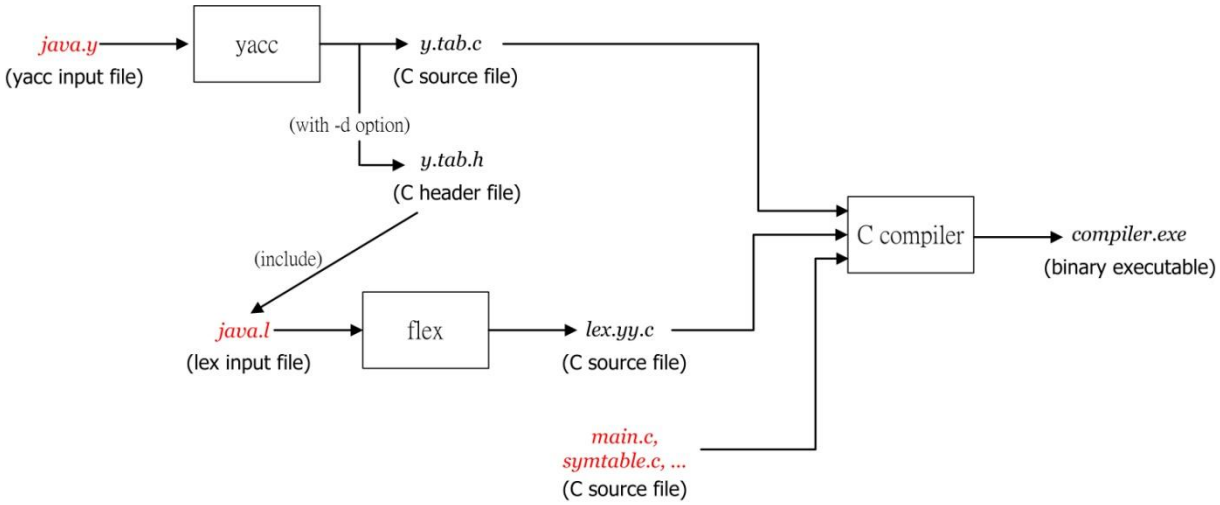
Le azioni nelle produzioni Yacc oltre a contenere specifiche routines che implementano la particolare traduzione input/output richiesta, servono soprattutto per calcolare $$ in funzione dei $i.

I valori degli attributi dei non-terminali sono computati dalle azioni semantiche, mentre i valori degli attributi dei token (vale a dire i lessemi) sono comunicati a Yacc dall'analizzatore lessicale attraverso la pseudo-variabile yylval.

Inoltre, se un'azione è inserita alla fine della parte destra di una produzione, allora essa sarà eseguita quando, durante il parsing, quella produzione sarà ridotta. Le azioni possono essere inserite anche all'interno della parte destra di produzioni; è chiaro che in questo caso nell'azione il contenuto di $$ può far riferimento solo ai $i dei simboli grammaticali alla sinistra dell'azione. Yacc ha un'azione di default che è { $$=$1;}. Pertanto se si scrive una produzione senza azioni semantiche, comunque Yacc esegue la sua azione di default quando il parser ridurrà quella produzione.

- Sezione **"ROUTINES AUSILIARIE"**

In questa sezione vi sono tutte le routines di supporto utili al corretto funzionamento del parser. In particolare le tre routines più importanti e che devono necessariamente essere presenti in questa sezione sono: l'analizzatore lessicale yylex(), la funzione main() e la funzione di gestione errori yyerror().



3.2 File Php-parser.y

```
%expect 2
```

Direttiva che imposta il numero di shift/reduce error che il parser si aspetta di trovare, e che non si è riusciti a risolvere in maniera predicibile.

```
%left T_INCLUDE T_EVAL T_REQUIRE

%left ','

%left T_LOGICAL_OR
```

```
%left T_LOGICAL_XOR

%left T_LOGICAL_AND

%right T_PRINT

%left '=' T_PLUS_EQUAL T_MINUS_EQUAL T_MUL_EQUAL T_DIV_EQUAL T_CONCAT_EQUAL
T_MOD_EQUAL T_AND_EQUAL T_OR_EQUAL T_XOR_EQUAL T_SL_EQUAL T_SR_EQUAL

%left '?' ':'

%left T_BOOLEAN_OR

%left T_BOOLEAN_AND

%left '|'

%left '^'

%left '&'

%nonassoc T_IS_EQUAL T_IS_NOT_EQUAL T_IS_IDENTICAL T_IS_NOT_IDENTICAL

%nonassoc '<' T_IS_SMALLER_OR_EQUAL '>' T_IS_GREATER_OR_EQUAL

%left T_SL T_SR
```

Si stabilisce l'associatività (destra o sinistra) dei vari operatori e tokens. In tal modo vengono eliminate eventuali ambiguità del linguaggio o possibilità di errori shift/reduce e reduce/reduce.

Gli operatori sono dichiarati in ordine crescente di precedenza. Tutti gli operatori dichiarati sulla stessa linea hanno lo stesso livello di precedenza.

```
%token T_LNUMBER

%token T_DNUMBER

%token T_STRING

%token T_VARIABLE

%token T_ENCAPSED_AND_WHITESPACE

%token T_CONSTANT_ENCAPSED_STRING

%token T_ECHO

%token T_DO

%token T_WHILE
```

In questo scorcio di codice vengono definiti i token multicarattere.

```
%union{ char *id;
        int numero;
}

%type <numero> parameter_list
%type <numero> function_call_parameter_list
```

In questa sezione viene dichiarata la union che servirà per definire i campi della variabile yylval, e viene definito il tipo dei due non terminali “parameter_list” e “function_call_parameter_list”, che serviranno al parser per tenere traccia del numero di parametri della funzione.

```
%start inner_statement_list
```

Direttiva che indica la regola di partenza, ovvero l'assioma della grammatica.

```
statement:
                '{' inner_statement_list '}'
        |       T_IF '(' expr ')' statement elseif_branch_list else_single
        |       T_IF '(' expr ')' ':' inner_statement_list new_elseif_branch_list new_else_single
T_ENDIF ';'
        |       T_IF expr ')' ':' inner_statement_list new_elseif_branch_list new_else_single
T_ENDIF ';' { yyerror("\033[01;31mERRORE SINTATTICO: '(' mancante nel costrutto IF\033[00m\n"); }
        |       T_WHILE '(' expr ')' while_statement
        |       T_WHILE '(' error ')' while_statement {yyerror("\033[01;31mERRORE SINTATTICO:
espressione nel costrutto WHILE non accettata\033[00m\n"); }
        |       T_WHILE expr ')' while_statement { yyerror("\033[01;31mERRORE SINTATTICO: '('
mancante nel costrutto WHILE\033[00m\n"); }
        |       T_DO statement T_WHILE '(' expr ')' ';'
        |       T_DO statement T_WHILE '(' expr ')' { yyerror("\033[01;31mERRORE SINTATTICO: ';'
mancante nel costrutto DO-WHILE\033[00m\n"); }
        |       T_FOR '(' for_expr_list ';' for_expr_list ';' for_expr_list ')' for_statement
        |       T_FOR for_expr_list ';' for_expr_list ';' for_expr_list ')' for_statement
{yyerror("\033[01;31mERRORE SINTATTICO: '(' mancante nel costrutto FOR\033[00m\n"); }
        |       T_FOR '(' error ';' for_expr_list ';' for_expr_list ')' for_statement
{yyerror("\033[01;31mERRORE SINTATTICO: primo argomento del costrutto FOR non
corretto\033[00m\n"); }
        |       T_FOR '(' for_expr_list ';' error ';' for_expr_list ')' for_statement
{yyerror("\033[01;31mERRORE SINTATTICO: secondo argomento del costrutto FOR non
corretto\033[00m\n"); }
```

```
        |       T_FOR '(' for_expr_list ';' for_expr_list ';' error ')' for_statement
{yyerror("\033[01;31mERRORE SINTATTICO: terzo argomento del costrutto FOR non
corretto\033[00m\n"); }
        |       T_SWITCH '(' expr ')' switch_case_list
        |       T_SWITCH expr ')' switch_case_list {yyerror("\033[01;31mERRORE SINTATTICO: '('
mancante nel costrutto SWITCH\033[00m\n"); }
        |       T_BREAK breakcontinue_expr ';'
        |       T_CONTINUE breakcontinue_expr ';'
        |       T_RETURN return_expr ';'
        |       T_GLOBAL global_var_list ';'
        |       T_STATIC static_var_list ';'
        |       T_ECHO echo_expr_list ';'
        |       T_ECHO error ';' {yyerror("\033[01;31mERRORE SINTATTICO: argomento della
funzione ECHO errato\033[00m\n");}
        |       expr ';'
;
```

Lista dei principali costrutti analizzati e definiti nella grammatica, con relativa gestione degli errori (come mostrato nel capitolo seguente).

```
%%

main(int ac, char **av) {

        extern FILE *yyin;

        tk = fopen("token_output.txt","w");

        fprintf(stream,"ELENCO PRODUZIONI UTILIZZATE:\n\n");
        fprintf(tk,"ELENCO TOKEN RICONOSCIUTI:\n\n");

        if(ac > 1 && (yyin = fopen(av[1], "r")) == NULL)
        {
                perror(av[1]);
                exit(1);
                fclose(tk);
        }
        printf("\033[01;32mANALISI DEL FILE: %s\033[00m\n\n", av[1]);

        if(!yyparse() && count == 0){ printf("\n\n\033[01;32mParsing PHP riuscito\033[00m\n");


        }else{ if(count!=0){printf("\n\n\033[01;31mParsing PHP fallito\033[00m\n");
```

```
                if(count>1)
                printf("\033[01;31mSono stati rilevati %d errori \033[00m\n",count);
                else printf("\033[01;31mE' stato rilevato %d errore \033[00m\n",count);
                segnalazione();
                }else printf("ERRORE FATALE\n");}

        fclose(tk);
}

yyerror (char *s)
{
if(s!="syntax error")
{
count++;
printf("\033[01;32mriga %2d: \033[00m",lineno);
printf("%s",s);}
}

int
yywrap(void) {
 return 1;
}
```

Nella terza sezione del file sono presenti funzioni e routines che saranno riportate nel sorgente finale, come ad esempio la gestione e l'implementazione della funzione built-in "yyerror" e il main del programma, oltre alla definizione della yywrap.

Nel main del programma viene aperto in scrittura un file di output, token_output.txt, che servirà a contenere tutti i token trovati.

La funzione yywrap() può essere usata per continuare a leggere da un ulteriore file. Viene richiamata alla fine di un file e si può allora aprire un altro file e restituire 0. Oppure si può restituire 1, indicando così che questa è effettivamente la fine di tutti i file di input, come nel nostro caso.

ANALISI ERRORI E SEMANTICA

4.1 Gestione e rilevamento errori

Nella realizzazione del progetto si è utilizzato un doppio approccio per il rilevamento degli errori.

Panic Mode : consente di non interrompre la fase di parsing quando vengono rilevati degli errori sintattici. Una volta incontrato l'errore infatti il parser riprende l'analisi in corrispondenza di alcuni token selezionati. In pratica si isola la frase che contiene l'errore riprendendo l'analisi della parte restante di codice non appena possibile.

Ad esempio nella seguente produzione:

```
statement ::= T_WHILE '(' error ')' while_statement
```

è stato utilizzato il terminale "error" disponibile in yacc. In questo modo quando l'argomento del while non risulta essere una espressione corretta viene matchata tale produzione, dove il terminale ')' fa da token di sincronizzazione, grazie al quale sarà quindi possibile riprendere l'analisi.

L'esempio seguente è quello relativo al costrutto for:

```
statement ::= T_FOR '(' error ';' for_expr_list ';' for_expr_list ')' for_statement
```

dove è stato utilizzato ancora il terminale "error" per sostituire una espressione di tipo for_expr_list non corretta. Questa volta il token di sincronizzazione è rappresentato dal ';' il quale rappresenta appunto il "confine" relativo all'input che non viene riconosciuto dalla grammatica. Viene prevista quindi la segnalazione di un errore sul primo degli argomenti del for. In maniera analoga si è sviluppato il riconoscimento di errori relativi agli altri parametri del for e di altri costrutti.

Error Production: consente anch'essa di evitare l'interruzione del parsing prevedendo l'utilizzo di produzioni che estendono la grammatica, in modo tale da generare degli errori comuni. Tale meccanismo risulta più statico e meno generico della tecnica Panic Mode, ma offre il vantaggio di essere piuttosto facile da implementare. Ad esempio nel codice si è inserita la produzione che prevede l'esecuzione della funzione yyerror() passando come argomento il messaggio di errore lessicale:

statement ::= T_DO statement T_WHILE '(' expr ')' { yyerror("ERRORE SINTATTICO");}

dove si è volutamente omesso il ';' per terminare correttamente il costrutto. Un altro esempio è relativo all'assenza di un termine in una produzione del tipo:

expr_without_variable ::= expr '&' expr

a questa viene quindi affiancata una produzione di questo tipo:

expr_without_variable ::= expr '&' { yyerror("ERRORE SINTATTICO");}

che ancora una volta richiama la funzione yyerror() per il passaggio del messaggio di errore riscontrato.

Passando al parser, questo codice errato:

```
1  <?php
2
3  var1 = 10;                              // errore: parte sinistra
4  $var2 = 8;                              // dell'espressione non riconosciuta
5
6  if( $var1 < $var2):
7  {
8        for(prova+er;$a<$var2;$a++)       // errore: primo argomento del
9             echo "messaggio!\n";         // costrutto for non accettato
10
11 }elseif($var1==$var2):
12       echo "nessun messaggio!\n"; */ // errore: chiusura commento
13 else:                                  // multiriga inatteso
14       {
15              echo "errore!\n";
16              while($var1 >= )          // errore: secondo termine
17                    $errori++;          // dell'espressione mancante
18       }
19 endif;                                 // errore: '(' mancante nel costrutto if
20
21 ?>
```

si ottiene il seguente output:

```
ANALISI DEL FILE: prova_sintattica1.php

riga  3: ERRORE SINTATTICO: parte sinistra dell'espressione non riconosciuta
riga  9: ERRORE SINTATTICO: primo argomento del costrutto FOR non corretto
riga 12: ERRORE SINTATTICO/LESSICALE: chiusura commento multiriga inaspettata
riga 16: ERRORE SINTATTICO: (>=) secondo termine dell'espressione mancante
riga 19: ERRORE SINTATTICO: '(' mancante nel costrutto IF


Parsing PHP fallito
Sono stati rilevati 5 errori
```

Si noti che per definire l'errore relativo alla chiusura inaspettata dei commenti multiriga è stato utilizzato un messaggio che evidenzia l'errore come "SINTATTICO/LESSICALE". Tale errore in realtà risulta essere sintattico in quanto è causato dal malposizionamento di determinati simboli, tuttavia si è specificato essere anche lessicale in quanto il riconoscimento avviene al livello dello scanner, il quale matcha i simboli "*/" di chiusura commenti quando la start condition <COMMENT> non risulta essere attiva.

A livello lessicale vengono riconosciuti due tipi di errori. Uno quando viene rilevata una variabile che comincia per un numero, l'altro quando viene matchato un carattere non corrispondente a nessuna delle forme previste dalle espressioni regolari già definite. Le regole per gestire tali errori sono le seguenti:

```
<IN_PHP>{T_VARIABLE_ERR}{
        yylval.id = (char *)strdup(yytext+2);
        yyerror("WARNING LESSICALE\n");
        return T_VARIABLE;
}

<IN_PHP>{T_VARIABLE_ERR2}{
        yyerror("ERRORE LESSICALE");
        return T_VARIABLE;
}
<IN_PHP,INITIAL,DOUBLE_QUOTES,BACKQUOTE,SINGLE_QUOTES,>.{
```

```
        yyerror("ERRORE LESSICALE");

}
```

dove sono stati definiti i seguenti alias:

```
T_VARIABLE_ERR "$"[0-9]{LABEL}

T_VARIABLE_ERR2 "$"[0-9][0-9]+{LABEL}

LABEL [a-zA-Z_\x7f-\xff][a-zA-Z0-9\x7f-\xff]*
```

La prima regola in realtà, come mostrato nel messaggio, non genera errore ma effettua una correzione automatica del nome di variabile, quindi salva le informazioni della variabile nella symbol table come se fosse scritta senza il numero (quindi correttamente) e restituisce il token T_VARIABLE.

La regola che invece rileva effettivamente un errore di tipo lessicale è la seconda, la quale non memorizza nessun valore all'interno della symbol table, cioè è come se non venisse trovata alcuna variabile ma viene comunque restituito il token T_VARIABLE per evitare che il parser si blocchi. Tali regole di gestione degli errori si basano sulla supposizione non deterministica, ma probabilmente lecita, che un utente che inserisca il simbolo '$' voglia fare riferimento ad una variabile, ed una variabile che comincia per un numero può derivare da una svista del programmatore e quindi corretta in automatico evitando il blocco delle fasi successive di compilazione, restituendo però come già detto un messaggio di warning.

Qui sotto è mostrato un esempio di codice lessicalmente errato:

```php
<?php

$2var1 = 10;            // warning: si sta correggendo in automatico il
$var2 = 5;              // nome della variabile eliminando il numero '2'

if($var1>$var2){
      echo "var1 maggiore di var2";
      $26var3 = $var1;  // errore: il nome di una variabile non può
}                       // cominciare con un numero

?>
```

Riportiamo qui di seguito l'output prodotto dal parsing del suddetto codice:

```
ANALISI DEL FILE: prova_lessicale1.php

riga  3: WARNING:e' stato corretto un errore lessicale
riga  8: ERRORE LESSICALE: variabile non può cominciare con numeri


Parsing PHP fallito
E' stato rilevato 1 errore
                  e 1 warning
```

Si noti come, nel caso in cui non ci fosse l'errore di riga 8 la compilazione potrebbe avvenire con successo nonostante l'errore commesso riguardo alla definizione di variabile di riga 8, avendo corretto la definizione stessa come precedentemente spiegato.

4.2 Semantica

Il sistema progettato, grazie allo yacc, consente un controllo di tipo semantico, ma solo per alcuni aspetti del linguaggio. L'analisi è stata rivolta essenzialmente alla definizione e alle chiamate di funzioni. Il meccanismo previsto consente di riconoscere se il nome di funzione che si sta cercando di definire non sia già stato utilizzato all'interno dello stesso programma php. Nel caso in cui tale verifica dia esito positivo (nome funzione già utilizzato) il parser rileverà un errore semantico specificando che la funzione non può essere ridefinita. In caso contrario (nome funzione non già utilizzato) si provvederà alla instanziazione e memorizzazione nella symbol table delle informazioni relative alla funzione.

Un altro errore semantico verrà rilevato nel caso in cui venga effettuata una chiamata ad una funzione che non è stata definita. Viene utilizzato un meccanismo di check che accedendo alla symbol table controlla l'esistenza della funzione che si sta cercando di richiamare e nel caso in cui questa non esiste verrà visualizzato un errore semantico relativo all'impossibilità di chiamare una funzione che non sia stata definita.

Infine è prevista un'ulteriore analisi di tipo semantico, sempre relativa alle funzioni, che viene svolta effettuando un controllo sul numero dei parametri che vengono passati durante la chiamata a funzione. Il sistema effettua un preventivo check relativo all'esistenza della funzione, poiché ovviamente si può effettuare una chiamata a funzione solo è stata preventivamente dichiarata. In seguito, se tale check ha avuto esito positivo, viene controllato che il numero di parametri della funzione chiamante coincida con il numero di parametri previsto nella dichiarazione della funzione stessa. Se questo numero non coincide verrà segnalato un errore semantico. Anche tale rilevazione sfrutta il meccanismo di consultazione della symbol table dove devono essere memorizzati, tra i vari valori, anche il numero di parametri della funzione, quando questa viene definita.

La symbol table è essenzialmente una linked lista atta a contenere i dati relativi a tutti gli elementi individuati durante la fase di parsing. Tali dati servono ad attribuire dei valori a delle variabili associate ai terminali ed ai non terminali della grammatica, tramite il quale si realizza appunto la descrizione di una grammatica ad attributi, che è essenziale per il riconoscimento semantico.

L'elemento basilare della lista ha la seguente struttura:

```
struct{

char *name

int numero_parametri;

struct symrec *next;}
```

Normalmente la tabella può prevedere un gran numero di parametri aggiuntivi, volti a particolareggiare i dati, in modo da approfondire l'analisi semantica. Ai fini del progetto si è comunque pensato di trascurare altri parametri e ci si è concentrati sugli attributi che servono a descrivere in chiave semantica le funzioni e le sue chiamate all'interno del codice.

Il primo campo della struttura è associato al nome della funzione. Il secondo campo è associato al numero di parametri che la funzione richiede le vengano passati, mentre il terzo ed ultimo parametro è utilizzato per realizzare la lista concatenata di elementi, esso rappresenta infatti il puntatore all'elemento successivo della lista.

Per la gestione di tale linked list sono previste delle funzioni putsym() e getsym() (descritte in appendice, nella libreria ST.h) che servono rispettivamente ad inserire dati nella lista e quindi nella symbol table e a richiamare dati dalla lista per effettuare controlli di vario tipo per verificare la correttezza semantica del codice.

Le succitate funzioni sono richiamate da altre funzioni che vengono a loro volta chiamate dal parser quando viene matchata una produzione cioè quando si ha una reduce ed in corrispondenza di questa è prevista un'azione semantica. Si usano a tale scopo le funzioni install() e context_check() (vedi appendice).

Qui di seguito è mostrato un semplice esempio di codice php contenente tre errori semantici delle tipologie sopra descritte.

```php
<?php

function dividi($a,$b)
{
     if($b!=0){
          $c=$a/$b;
          return $c;
          }else return "error";
}

function dividi($n,$d){}  //errore: impossibile ridefinire  dividi()

$out = divido(10,2);      //errore: funzione non definita
echo $out;

$out = dividi(10,2,1);    //errore: numero dei parametri non corretto

?>
```

Passando il testo sopra mostrato si avrà in output il seguente risultato:

```
ANALISI DEL FILE: prova_semantica1.php

riga 11: ERRORE SEMANTICO: impossibile ridefinire la funzione dividi()
riga 13: ERRORE SEMANTICO: chiamata a funzione non definita
riga 16: ERRORE SEMANTICO: numero parametri della chiamata a funzione non corretto


Parsing PHP fallito
Sono stati rilevati 3 errori
```

BIBLIOGRAFIA

Lex & Yacc - John R. Levine, Tony Mason, Doug Brown

Compilers: Principles, Techniques and Tools - Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman

Lucidi e materiale del docente

http://it.wikipedia.org/wiki/PHP

http://www.devincook.com/goldparser

http://it.php.net/manual/en/tokens.php

http://dinosaur.compilertools.net

APPENDICE

1. Php-parser.y

```
      /*

            Grammatica PHP 5.2.16 per l'esame di Compilatori ed
Interpreti
                        Autori: M. Rinaldi & G. Villani
       */

%{

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "ST.h"
int errors=0;
int param=0;
int parTot;
FILE *tk;
int nprod=0;
int lineno;

void install(char *sym_name,int num_par)
{
      symrec *s;
      s=getsym(sym_name,num_par,0);

      if(s==0)
            {s=putsym(sym_name,num_par);}
      else{
            yyerror("\033[01;34mERRORE SEMANTICO: impossibile
ridefinire la funzione \033[00m");
            printf("\033[01;34m%s()\033[00m\n",sym_name);
      }
}

context_check(char *sym_name,int numero_parametri)
{


      if(getsym(sym_name,numero_parametri,1)==0)
            yyerror("\033[01;34mERRORE SEMANTICO: chiamata a funzione
non definita\033[00m\n");

}

int count = 0;

%}
```

```
%expect 2


%left T_INCLUDE T_EVAL T_REQUIRE
%left ','
%left T_LOGICAL_OR
%left T_LOGICAL_XOR
%left T_LOGICAL_AND
%right T_PRINT
%left '=' T_PLUS_EQUAL T_MINUS_EQUAL T_MUL_EQUAL T_DIV_EQUAL
T_CONCAT_EQUAL T_MOD_EQUAL T_AND_EQUAL T_OR_EQUAL T_XOR_EQUAL
T_SL_EQUAL T_SR_EQUAL
%left '?' ':'
%left T_BOOLEAN_OR
%left T_BOOLEAN_AND
%left '|'
%left '^'
%left '&'
%nonassoc T_IS_EQUAL T_IS_NOT_EQUAL T_IS_IDENTICAL
T_IS_NOT_IDENTICAL
%nonassoc '<' T_IS_SMALLER_OR_EQUAL '>' T_IS_GREATER_OR_EQUAL
%left T_SL T_SR
%left '+' '-' '.'
%left '*' '/' '%'
%right '!'
%right '~' T_INC T_DEC T_INT_CAST T_DOUBLE_CAST T_STRING_CAST
T_BOOL_CAST '@'
%token T_EXIT
%token T_IF
%left T_ELSEIF
%left T_ELSE
%left T_ENDIF
%left ')'
%token T_LNUMBER
%token T_DNUMBER
%token <id> T_STRING
%token T_VARIABLE
%token T_ENCAPSED_AND_WHITESPACE
%token T_CONSTANT_ENCAPSED_STRING
%token T_ECHO
%token T_DO
%token T_WHILE
%token T_ENDWHILE
%token T_FOR
%token T_ENDFOR
%token T_SWITCH
%token T_ENDSWITCH
%token T_CASE
%token T_DEFAULT
%token T_BREAK
%token T_CONTINUE
%token T_FUNCTION
%token T_RETURN
%token T_GLOBAL
%right T_STATIC
%token T_ISSET
```

```
%token T_EMPTY
%token T_LINE
%token T_FILE

%union{ char *id;
     int numero;
}

%type <numero> parameter_list
%type <numero> function_call_parameter_list

%start inner_statement_list

%% /* Regole */

inner_statement_list:
          inner_statement_list inner_statement
     |    /* empty */
;

inner_statement:
          statement
     |    function_declaration_statement
;



statement:
          '{' inner_statement_list '}'
     |    T_IF '(' expr ')' statement elseif_branch_list
else_single
     |    T_IF '(' expr ')' ':' inner_statement_list
new_elseif_branch_list new_else_single T_ENDIF ';'
     |    T_IF expr ')' ':' inner_statement_list
new_elseif_branch_list new_else_single T_ENDIF ';' {
yyerror("\033[01;31mERRORE SINTATTICO: '(' mancante nel costrutto
IF\033[00m\n");  }
     |    T_WHILE '(' expr ')' while_statement
     |    T_WHILE '(' error ')' while_statement
{yyerror("\033[01;31mERRORE SINTATTICO: espressione nel costrutto
WHILE non accettata\033[00m\n");  }
     |    T_WHILE expr ')' while_statement {
yyerror("\033[01;31mERRORE SINTATTICO: '(' mancante nel costrutto
WHILE\033[00m\n"); }
     |    T_DO statement T_WHILE '(' expr ')' ';'
     |    T_DO statement T_WHILE '(' expr ')' {
yyerror("\033[01;31mERRORE SINTATTICO: ';' mancante nel costrutto
DO-WHILE\033[00m\n"); }
     |    T_FOR '(' for_expr_list ';' for_expr_list ';'
for_expr_list ')' for_statement
     |    T_FOR for_expr_list ';' for_expr_list ';' for_expr_list
')' for_statement {yyerror("\033[01;31mERRORE SINTATTICO: '('
mancante nel costrutto FOR\033[00m\n"); }
     |    T_FOR '(' error ';' for_expr_list ';' for_expr_list ')'
for_statement {yyerror("\033[01;31mERRORE SINTATTICO: primo
argomento del costrutto FOR non corretto\033[00m\n"); }
```

```
     |    T_FOR '(' for_expr_list ';' error ';' for_expr_list ')'
for_statement {yyerror("\033[01;31mERRORE SINTATTICO: secondo
argomento del costrutto FOR non corretto\033[00m\n"); }
     |    T_FOR '(' for_expr_list ';' for_expr_list ';' error ')'
for_statement {yyerror("\033[01;31mERRORE SINTATTICO: terzo
argomento del costrutto FOR non corretto\033[00m\n"); }
     |    T_SWITCH '(' expr ')' switch_case_list
     |    T_SWITCH expr ')' switch_case_list
{yyerror("\033[01;31mERRORE SINTATTICO: '(' mancante nel costrutto
SWITCH\033[00m\n"); }
     |    T_BREAK breakcontinue_expr ';'
     |    T_CONTINUE breakcontinue_expr ';'
     |    T_RETURN return_expr ';'
     |    T_GLOBAL global_var_list ';'
     |    T_STATIC static_var_list ';'
     |    T_ECHO echo_expr_list ';'
     |    T_ECHO error ';' {yyerror("\033[01;31mERRORE SINTATTICO:
argomento della funzione ECHO errato\033[00m\n");}
     |    expr ';'
;

function_declaration_statement:
          T_FUNCTION T_STRING '(' parameter_list ')' '{'
inner_statement_list '}' { install($2,$4);}
     |    T_FUNCTION '&' T_STRING '(' parameter_list ')' '{'
inner_statement_list '}'
;

parameter_list:
          nonempty_parameter_list {$$=parTot; }
     |    /* empty */ {$$=0;}
;

nonempty_parameter_list:
          parameter ',' nonempty_parameter_list { param++;
parTot=param;}
     |    parameter { param=1; }
;

parameter:
          '&' T_VARIABLE
     |    '&' T_VARIABLE '=' static_scalar
     |    T_VARIABLE
     |    T_VARIABLE '=' static_scalar
     |    T_STRING {yyerror("\033[01;31mERRORE SINTATTICO: il
parametro della definizione di funzione non può essere una
stringa\033[00m\n"); }
     |    T_CONSTANT_ENCAPSED_STRING {yyerror("\033[01;31mERRORE
SINTATTICO: il parametro di una dichiarazione di funzione non può
essere una stringa\033[00m\n");}
     |    T_LNUMBER {yyerror("\033[01;31mERRORE SINTATTICO: il
parametro di una dichiarazione di funzione non può essere un
numero\033[00m\n"); }
     |    T_DNUMBER {yyerror("\033[01;31mERRORE SINTATTICO: il
parametro di una dichiarazione di funzione non può essere un
numero\033[00m\n"); }
```

```
;

expr:
          variable
     |    expr_without_variable
;

variable:
          reference_variable
     |    function_call
;

function_call:
          T_STRING '(' function_call_parameter_list ')'
{context_check($1,$3);}
;

function_call_parameter_list:
          nonempty_function_call_parameter_list {$$=parTot; }
     |    /* empty */ {$$=0; }
;

nonempty_function_call_parameter_list:
          expr_without_variable {param=1; }
     |    variable {param=1; }
     |    '&' variable {param=1; }
     |    nonempty_function_call_parameter_list ','
expr_without_variable {param++; parTot=param; }
     |    nonempty_function_call_parameter_list ',' variable
{param++; parTot=param; }
     |    nonempty_function_call_parameter_list ',' '&' variable
{param++; parTot=param; }
;

reference_variable:
          compound_variable
     |    reference_variable '{' expr '}'
;

compound_variable:
          T_VARIABLE
     |    '$' '{' expr '}'
;

elseif_branch:
          T_ELSEIF '(' expr ')' statement
;

elseif_branch_list:
          elseif_branch_list elseif_branch
     |    /* empty */
;

else_single:
          T_ELSE statement
     |    /* empty */
```

```
;

new_elseif_branch:
          T_ELSEIF '(' expr ')' ':' inner_statement_list
     |    T_ELSEIF '(' expr ')' inner_statement_list
{yyerror("\033[01;31mERRORE SINTATTICO: ':' mancanti nel costrutto
ELSEIF \033[00m\n"); }
;

new_elseif_branch_list:
          new_elseif_branch_list new_elseif_branch
     |    /* empty */
;

new_else_single:
          T_ELSE ':' inner_statement_list
     |    /* empty */
;

while_statement:
          statement
     |    ':' inner_statement_list T_ENDWHILE ';'
     |    ':' inner_statement_list T_ENDWHILE
{yyerror("\033[01;31mERRORE SINTATTICO: ';' mancante nel costrutto
WHILE\033[00m\n"); }
;

for_expr_list:
          nonempty_for_expr
     |    /* empty */
;

nonempty_for_expr:
          nonempty_for_expr ',' expr
     |    expr
;

for_statement:
          statement
     |    ':' inner_statement_list T_ENDFOR ';'
;

switch_case_list:
          '{' case_list '}'
     |    '{' ';' case_list '}'
     |    ':' case_list T_ENDSWITCH ';'
     |    ':' ';' case_list T_ENDSWITCH ';'
;

case_list:
          case_list T_CASE expr ':' inner_statement_list
     |    case_list T_CASE expr ';' inner_statement_list
     |    case_list T_DEFAULT ':' inner_statement_list
     |    case_list T_DEFAULT ';' inner_statement_list
     |    /* empty */
;
```

```
breakcontinue_expr:
          expr
     |    /* empty */
;

return_expr:
          expr_without_variable
     |    variable
     |    /* empty */
;

expr_without_variable:
          variable '=' expr
     |    error '=' expr {yyerror("\033[01;31mERRORE SINTATTICO:
parte sinistra dell'espressione non riconosciuta \033[00m\n"); }
     |    variable '=' '&' variable
     |    variable T_PLUS_EQUAL expr
     |    error T_PLUS_EQUAL expr {yyerror("\033[01;31mERRORE
SINTATTICO: parte sinistra dell'espressione non riconosciuta
\033[00m\n"); }
     |    variable T_MINUS_EQUAL expr
     |    variable T_MUL_EQUAL expr
     |    variable T_DIV_EQUAL expr
     |    variable T_CONCAT_EQUAL expr
     |    variable T_MOD_EQUAL expr
     |    variable T_AND_EQUAL expr
     |    variable T_OR_EQUAL expr
     |    variable T_XOR_EQUAL expr
     |    variable T_SL_EQUAL expr
     |    variable T_SR_EQUAL expr
     |    variable T_INC
     |    T_INC variable
     |    variable T_DEC
     |    T_DEC variable
     |    expr T_BOOLEAN_OR expr
     |    expr T_BOOLEAN_OR {yyerror("\033[01;31mERRORE SINTATTICO:
(||) secondo termine dell'espressione mancante\033[00m\n"); }
     |    expr T_BOOLEAN_AND expr
     |    expr T_BOOLEAN_AND {yyerror("\033[01;31mERRORE
SINTATTICO: (&&) secondo termine dell'espressione
mancante\033[00m\n"); }
     |    expr T_LOGICAL_OR expr
     |    expr T_LOGICAL_OR {yyerror("\033[01;31mERRORE SINTATTICO:
(OR) secondo termine dell'espressione mancante\033[00m\n"); }
     |    expr T_LOGICAL_AND expr
     |    expr T_LOGICAL_AND {yyerror("\033[01;31mERRORE
SINTATTICO: (AND) secondo termine dell'espressione
mancante\033[00m\n"); }
     |    expr T_LOGICAL_XOR expr
     |    expr T_LOGICAL_XOR {yyerror("\033[01;31mERRORE
SINTATTICO: (XOR) secondo termine dell'espressione
mancante\033[00m\n"); }
     |    expr '|' expr
     |    expr '|' {yyerror("\033[01;31mERRORE SINTATTICO: (|)
secondo termine dell'espressione mancante\033[00m\n"); }
```

```
      |     expr '&' expr
      |     expr '&' { yyerror("\033[01;31mERRORE SINTATTICO: (&)
secondo termine dell'espressione mancante\033[00m\n"); }
      |     expr '^' expr
      |     expr '^' { yyerror("\033[01;31mERRORE SINTATTICO: (^)
secondo termine dell'espressione mancante\033[00m\n"); }
      |     expr '.' expr
      |     expr '.' { yyerror("\033[01;31mERRORE SINTATTICO: (.)
secondo termine dell'espressione mancante\033[00m\n"); }
      |     expr '+' expr
      |     expr '+' { yyerror("\033[01;31mERRORE SINTATTICO: (+)
secondo termine dell'espressione mancante\033[00m\n"); }
      |     expr '-' expr
      |     expr '-' { yyerror("\033[01;31mERRORE SINTATTICO: (-)
secondo termine dell'espressione mancante\033[00m\n"); }
      |     expr '*' expr
      |     expr '*' { yyerror("\033[01;31mERRORE SINTATTICO: (*)
secondo termine dell'espressione mancante\033[00m\n"); }
      |     expr '/' expr
      |     expr '/' { yyerror("\033[01;31mERRORE SINTATTICO: (/)
secondo termine dell'espressione mancante\033[00m\n"); }
      |     expr '%' expr
      |     expr '%' { yyerror("\033[01;31mERRORE SINTATTICO: (%)
secondo termine dell'espressione mancante\033[00m\n"); }
      |     expr T_SL expr
      |     expr T_SL { yyerror("\033[01;31mERRORE SINTATTICO: (<<)
secondo termine dell'espressione mancante\033[00m\n"); }
      |     expr T_SR expr
      |     expr T_SR { yyerror("\033[01;31mERRORE SINTATTICO: (>>)
secondo termine dell'espressione mancante\033[00m\n"); }
      |     expr T_IS_IDENTICAL expr
      |     expr T_IS_IDENTICAL { yyerror("\033[01;31mERRORE
SINTATTICO: (===) secondo termine dell'espressione
mancante\033[00m\n"); }
      |     expr T_IS_EQUAL expr
      |     expr T_IS_EQUAL { yyerror("\033[01;31mERRORE SINTATTICO:
(==) secondo termine dell'espressione mancante\033[00m\n"); }
      |     expr T_IS_NOT_EQUAL expr
      |     expr T_IS_NOT_EQUAL { yyerror("\033[01;31mERRORE
SINTATTICO: (!=) secondo termine dell'espressione
mancante\033[00m\n"); }
      |     expr T_IS_NOT_IDENTICAL expr
      |     expr T_IS_NOT_IDENTICAL { yyerror("\033[01;31mERRORE
SINTATTICO: (!==) secondo termine dell'espressione
mancante\033[00m\n"); }
      |     expr '<' expr
      |     expr '<' { yyerror("\033[01;31mERRORE SINTATTICO: (<)
secondo termine dell'espressione mancante\033[00m\n"); }
      |     expr T_IS_SMALLER_OR_EQUAL expr
      |     expr T_IS_SMALLER_OR_EQUAL { yyerror("\033[01;31mERRORE
SINTATTICO: (<=) secondo termine dell'espressione
mancante\033[00m\n"); }
      |     expr '>' expr
      |     expr '>' { yyerror("\033[01;31mERRORE SINTATTICO: (>)
secondo termine dell'espressione mancante\033[00m\n"); }
      |     expr T_IS_GREATER_OR_EQUAL expr
```

```
      |     expr T_IS_GREATER_OR_EQUAL { yyerror("\033[01;31mERRORE
SINTATTICO: (>=) secondo termine dell'espressione
mancante\033[00m\n"); }
      |     '+' expr
      |     '-' expr
      |     '!' expr
      |     '~' expr
      |     '(' expr ')'
      |     expr '?' expr ':' expr
      |     internal_functions
      |     T_INT_CAST expr
      |     T_DOUBLE_CAST expr
      |     T_STRING_CAST expr
      |     T_BOOL_CAST expr
      |     T_EXIT exit_expr
      |     '@' expr
      |     scalar
      |     T_PRINT expr
;

exit_expr:
            /* empty */
      |     '(' ')'
      |     '(' expr ')'
;

variable_list:
            variable_list ',' variable
      |     variable
;

internal_functions:
            T_ISSET '(' variable_list ')'
      |     T_ISSET variable_list ')' {yyerror("\033[01;31mERRORE
SINTATTICO: '(' mancante dopo la dichiarazione della funzione
isset()\033[00m\n");}
      |     T_EMPTY '(' variable ')'
      |     T_EMPTY variable ')' {yyerror("\033[01;31mERRORE
SINTATTICO: ')' mancante dopo la dichiarazione della funzione
empty()\033[00m\n");}
      |     T_INCLUDE expr
      |     T_EVAL '(' expr ')'
      |     T_REQUIRE expr
;

scalar:
            common_scalar
;

common_scalar:
            T_LNUMBER
      |     T_DNUMBER
      |     T_CONSTANT_ENCAPSED_STRING
      |     T_LINE
      |     T_FILE
      |     T_ENCAPSED_AND_WHITESPACE
```

```
;

static_scalar:
          common_scalar
     |    '+' static_scalar
     |    '-' static_scalar
;


global_var_list:
          global_var_list ',' global_var
     |    global_var
;

global_var:
          T_VARIABLE
     |    '$' variable
     |    '$''{' expr '}'
;

static_var_list:
          static_var_list ',' static_var
     |    static_var
;

static_var:
          T_VARIABLE
     |    T_VARIABLE '=' static_scalar
;

echo_expr_list:
          echo_expr_list ',' expr
     |    expr
;

%%


main(int ac, char **av) {

     extern FILE *yyin;

     tk = fopen("token_output.txt","w");

     fprintf(tk,"ELENCO TOKEN RICONOSCIUTI:\n\n");

     if(ac > 1 && (yyin = fopen(av[1], "r")) == NULL)
     {
          perror(av[1]);
          exit(1);
          fclose(tk);
     }
     printf("\033[01;32mANALISI DEL FILE: %s\033[00m\n\n", av[1]);

     if(!yyparse() && count == 0){ printf("\n\n\033[01;32mParsing
PHP riuscito\033[00m\n");
```

```
     }else{ if(count!=0){printf("\n\n\033[01;31mParsing PHP
fallito\033[00m\n");
          if(count>1)
          printf("\033[01;31mSono stati rilevati %d errori
\033[00m\n",count);
          else printf("\033[01;31mE' stato rilevato %d errore
\033[00m\n",count);
          segnalazione();
          }else printf("\033[01;31mERRORE FATALE: impossibile
procedere con la compilazione\033[00m\n");}

     fclose(tk);

}

yyerror (char *s)
{

if(s!="syntax error")
{
count++;
printf("\033[01;32mriga %2d: \033[00m",lineno);
printf("%s",s);}
}

int
yywrap(void) {
  return 1;
}
```

2. Php-scanner.l

```
%{

      /*
              SCANNER per il linguaggio PHP
            Autori: M. Rinaldi & G. Villani
      */

%}

%x IN_PHP
%x DOUBLE_QUOTES
%x SINGLE_QUOTE
%x BACKQUOTE
%x COMMENT

%{

#include <stdio.h>
#include "y.tab.h"

int warn=0;
int lineno = 1;
void yyerror(char *s);
FILE *tk;
int count;

%}


LNUM [0-9]+
DNUM ([0-9]*[\.][0-9]+)|([0-9]+[\.][0-9]*)
HEXNUM "0x"[0-9a-fA-F]+
EXPONENT_DNUM (({LNUM}|{DNUM})[eE][+-]?{LNUM})

LABEL[a-zA-Z_\x7f-\xff][a-zA-Z0-9\x7f-\xff]*
T_STRING {LABEL}
T_VARIABLE "$"{LABEL}
T_VARIABLE_ERR "$"[0-9]{LABEL}
T_VARIABLE_ERR2 "$"[0-9][0-9]+{LABEL}

WHITESPACE [ \n\r\t]+
FORM [\\][nrt]
TABS_AND_SPACES [ \t]*
TOKENS [-;:,.\[\](|^&+/*=%!~<>{}?@]
ENCAPSED_TOKENS [\[\]{}$]
ESCAPED_AND_WHITESPACE [-\n\t\r ]+

%%

<IN_PHP>"exit" {
      fprintf(tk,"T_EXIT\n");
      return T_EXIT;
```

```
}

<IN_PHP>"die" {
      fprintf(tk,"T_DIE\n");
      return T_EXIT;
}

<IN_PHP>"function"|"cfunction" {
      fprintf(tk,"T_FUNCTION\n");
      return T_FUNCTION;
}

<IN_PHP>"return" {
      fprintf(tk,"T_RETURN\n");
      return T_RETURN;
}

<IN_PHP>"if" {
      fprintf(tk,"T_IF\n");
      return T_IF;
}

<IN_PHP>"elseif" {
      fprintf(tk,"T_ELSEIF\n");
      return T_ELSEIF;
}

<IN_PHP>"endif" {
      fprintf(tk,"T_ENDIF\n");
      return T_ENDIF;
}

<IN_PHP>"else" {
      fprintf(tk,"T_ELSE\n");
      return T_ELSE;
}

<IN_PHP>"while" {
      fprintf(tk,"T_WHILE\n");
      return T_WHILE;
}

<IN_PHP>"endwhile" {
      fprintf(tk,"T_ENDWHILE\n");
      return T_ENDWHILE;
}

<IN_PHP>"do" {
      fprintf(tk,"T_DO\n");
      return T_DO;
}

<IN_PHP>"for" {
      fprintf(tk,"T_FOR\n");
      return T_FOR;
}
```

```
<IN_PHP>"endfor" {
      fprintf(tk,"T_ENDFOR\n");
      return T_ENDFOR;
}

<IN_PHP>"switch" {
      fprintf(tk,"T_SWITCH\n");
      return T_SWITCH;
}

<IN_PHP>"endswitch" {
      fprintf(tk,"T_ENDSWITCH\n");
      return T_ENDSWITCH;
}

<IN_PHP>"case" {
      fprintf(tk,"T_CASE\n");
      return T_CASE;
}

<IN_PHP>"default" {
      fprintf(tk,"T_DEFAULT\n");
      return T_DEFAULT;
}

<IN_PHP>"break" {
      fprintf(tk,"T_BREAK\n");
      return T_BREAK;
}

<IN_PHP>"continue" {
      fprintf(tk,"T_CONTINUE\n");
      return T_CONTINUE;
}

<IN_PHP>"echo" {
      fprintf(tk,"T_ECHO\n");
      return T_ECHO;
}

<IN_PHP>"print" {
      fprintf(tk,"T_PRINT\n");
      return T_PRINT;
}

<IN_PHP>"("{TABS_AND_SPACES}("int"|"integer"){TABS_AND_SPACES}")" {
      fprintf(tk,"T_INT_CAST\n");
      return T_INT_CAST;
}

<IN_PHP>"("{TABS_AND_SPACES}("real"|"double"|"float"){TABS_AND_SPACE
S}")" {
      fprintf(tk,"T_DOUBLE_CAST\n");
      return T_DOUBLE_CAST;
}
```

```
<IN_PHP>"("{TABS_AND_SPACES}"string"{TABS_AND_SPACES}")" {
      fprintf(tk,"T_STRING_CAST\n");
      return T_STRING_CAST;
}

<IN_PHP>"("{TABS_AND_SPACES}("bool"|"boolean"){TABS_AND_SPACES}")" {
      fprintf(tk,"T_BOOL_CAST\n");
      return T_BOOL_CAST;
}

<IN_PHP>"eval" {
      fprintf(tk,"T_EVAL\n");
      return T_EVAL;
}

<IN_PHP>"include" {
      fprintf(tk,"T_INCLUDE\n");
      return T_INCLUDE;
}

<IN_PHP>"require" {
      fprintf(tk,"T_ELSE\n");
      return T_REQUIRE;
}

<IN_PHP>"global" {
      fprintf(tk,"T_GLOBAL\n");
      return T_GLOBAL;
}

<IN_PHP>"isset" {
      fprintf(tk,"T_ISSET\n");
      return T_ISSET;
}

<IN_PHP>"empty" {
      fprintf(tk,"T_EMPTY\n");
      return T_EMPTY;
}

<IN_PHP>"static" {
      fprintf(tk,"T_STATIC\n");
      return T_STATIC;
}

<IN_PHP>"++" {
      fprintf(tk,"T_INC\n");
      return T_INC;
}

<IN_PHP>"--" {
      fprintf(tk,"T_DEC\n");
      return T_DEC;
}
```

```
<IN_PHP>"==" {
      fprintf(tk,"T_IS_EQUAL\n");
      return T_IS_EQUAL;
}

<IN_PHP>"===" {
      fprintf(tk,"T_IS_IDENTICAL\n");
      return T_IS_IDENTICAL;
}

<IN_PHP>"!==" {
      fprintf(tk,"T_IS_NOT_IDENTICAL\n");
      return T_IS_NOT_IDENTICAL;
}

<IN_PHP>"!="|"<>" {
      fprintf(tk,"T_IS_NOT_EQUAL\n");
      return T_IS_NOT_EQUAL;
}

<IN_PHP>"<=" {
      fprintf(tk,"T_IS_SMALLER_OR_EQUAL\n");
      return T_IS_SMALLER_OR_EQUAL;
}

<IN_PHP>">=" {
      fprintf(tk,"T_IS_GREATER_OR_EQUAL\n");
      return T_IS_GREATER_OR_EQUAL;
}

<IN_PHP>"+=" {
      fprintf(tk,"T_PLUS_EQUAL\n");
      return T_PLUS_EQUAL;
}

<IN_PHP>"-=" {
      fprintf(tk,"T_MINUS_EQUAL\n");
      return T_MINUS_EQUAL;
}

<IN_PHP>"*=" {
      fprintf(tk,"T_MUL_EQUAL\n");
      return T_MUL_EQUAL;
}

<IN_PHP>"/=" {
      fprintf(tk,"T_DIV_EQUAL\n");
      return T_DIV_EQUAL;
}

<IN_PHP>".=" {
      fprintf(tk,"T_CONCAT_EQUAL\n");
      return T_CONCAT_EQUAL;
}
```

```
<IN_PHP>"%=" {
      fprintf(tk,"T_MOD_EQUAL\n");
      return T_MOD_EQUAL;
}

<IN_PHP>"<<=" {
      fprintf(tk,"T_SL_EQUAL\n");
      return T_SL_EQUAL;
}

<IN_PHP>">>=" {
      fprintf(tk,"T_SR_EQUAL\n");
      return T_SR_EQUAL;
}

<IN_PHP>"&=" {
      fprintf(tk,"T_AND_EQUAL\n");
      return T_AND_EQUAL;
}

<IN_PHP>"|=" {
      fprintf(tk,"T_OR_EQUAL\n");
      return T_OR_EQUAL;
}

<IN_PHP>"^=" {
      fprintf(tk,"T_XOR_EQUAL\n");
      return T_XOR_EQUAL;
}

<IN_PHP>"||" {
      fprintf(tk,"T_BOOLEAN_OR\n");
      return T_BOOLEAN_OR;
}

<IN_PHP>"&&" {
      fprintf(tk,"T_BOOLEAN_AND\n");
      return T_BOOLEAN_AND;
}

<IN_PHP>"OR" {
      fprintf(tk,"T_LOGICAL_OR\n");
      return T_LOGICAL_OR;
}

<IN_PHP>"AND" {
      fprintf(tk,"T_LOGICAL_AND\n");
      return T_LOGICAL_AND;
}

<IN_PHP>"XOR" {
      fprintf(tk,"T_LOGICAL_XOR\n");
      return T_LOGICAL_XOR;
}

<IN_PHP>"<<" {
```

```
      fprintf(tk,"T_SL\n");
      return T_SL;
}

<IN_PHP>">>" {
      fprintf(tk,"T_SR\n");
      return T_SR;
}


<IN_PHP>{TOKENS} {
      fprintf(tk,"%c\n",yytext[0]);
      return yytext[0];
}

<IN_PHP>{T_VARIABLE} {
      fprintf(tk,"T_VARIABLE\n");
      yylval.id = (char *)strdup(yytext+1);
      return T_VARIABLE;
}

<IN_PHP>{T_VARIABLE_ERR} {
      fprintf(tk,"T_VARIABLE\n");
      yylval.id = (char *)strdup(yytext+2);
      yyerror("\033[01;33mWARNING:e' stato corretto un errore
lessicale\033[00m\n");
      count--;
      warn++;
      return T_VARIABLE;
}

<IN_PHP>{T_VARIABLE_ERR2} {
      yyerror("\033[01;33mERRORE LESSICALE: variabile non puÃ²
cominciare con numeri\033[00m\n");
      return T_VARIABLE;

}

<IN_PHP>{LABEL} {
      fprintf(tk,"T_STRING\n");
      yylval.id = (char *)strdup(yytext);
      return T_STRING;
}

<IN_PHP>{LNUM}|{HEXNUM} {
      fprintf(tk,"T_LNUMBER\n");
      yylval.numero=atoi(yytext);
      return T_LNUMBER;
}

<IN_PHP>{DNUM}|{EXPONENT_DNUM} {
      fprintf(tk,"T_DNUMBER\n");
      yylval.numero=atof(yytext);
      return T_DNUMBER;
}
```

```
<IN_PHP>"__LINE__" {
      fprintf(tk,"T_LINE\n");
      return T_LINE;
}

<IN_PHP>"__FILE__" {
      fprintf(tk,"T_FILE\n");
      return T_FILE;
}


<INITIAL>\n {
      lineno++;
}

<INITIAL>"<?"|"<script"{WHITESPACE}+"language"{WHITESPACE}*"="{WHITE
SPACE}*("php"|"\"php\""|"\'php\'"){WHITESPACE}*">" {
      BEGIN(IN_PHP);
      //printf("Inizio script PHP\n");
}



<INITIAL>"<?php"[ \n\r\t] {
      if (yytext[yyleng-1] == '\n') {
            lineno++;
      }
      BEGIN(IN_PHP);
      //printf("Inizio script PHP\n");
}

<IN_PHP>("?>"|"</script"{WHITESPACE}*">")([\n]|"\r\n")? {
      BEGIN(INITIAL);
      //printf("fine script PHP\n\nRighe:%d\n",lineno);
}

<IN_PHP>["] {
      BEGIN(DOUBLE_QUOTES);

}

<IN_PHP>[`] {
      BEGIN(BACKQUOTE);

}


<IN_PHP>['] {
      BEGIN(SINGLE_QUOTE);

}

<DOUBLE_QUOTES>["] {
      BEGIN(IN_PHP);

}
```

```
<BACKQUOTE>[`] {
      BEGIN(IN_PHP);

}


<SINGLE_QUOTE>['] {
      BEGIN(IN_PHP);

}

<DOUBLE_QUOTES,SINGLE_QUOTE,BACKQUOTE>{LABEL}({WHITESPACE}|{FORM}|{L
ABEL}|{LNUM}|{TOKENS})* {
      fprintf(tk,"T_CONSTANT_ENCAPSED_STRING\n");
      return T_CONSTANT_ENCAPSED_STRING;
}

<DOUBLE_QUOTES,BACKQUOTE>{ESCAPED_AND_WHITESPACE} {
      fprintf(tk,"T_ENCAPSED_AND_WHITESPACE\n");
      return T_ENCAPSED_AND_WHITESPACE;
}

<IN_PHP>([#]|"//")([^\n\r?]|"?"[^>\n\r])*("?\n"|"?\r\n")? {
      if (yytext[yyleng-1] == '\n') {
            lineno++;
      }
      //printf("Commento eliminato con doppio slash o
cancelletto\n");
}

<IN_PHP>^[ \t]*"/*" {
      BEGIN(COMMENT);
      //printf("Inizio commento multiriga\n");
}

<IN_PHP>^[ \t]*"/*".*"*/"[ \t]*\n {
      if (yytext[yyleng-1] == '\n') {
            lineno++;
      }
      //printf("Commento su singola linea eliminato\n");
}

<IN_PHP>"*/" { yyerror("\033[01;33mERRORE SINTATTICO/LESSICALE:
chiusura commento multiriga inaspettata\033[00m\n");}


<COMMENT>"*/"[ \t]*\n {
      BEGIN(IN_PHP);
      //printf("Commento multiriga eliminato\n");
      if (yytext[yyleng-1] == '\n') {
                  lineno++;
            }
}
```

```
<COMMENT>"*/" {
      BEGIN(IN_PHP);
      //printf("Fine commento\n");
}

<COMMENT>\n {
      if (yytext[yyleng-1] == '\n') {
                  lineno++;
            }
}

<COMMENT>[a-zA-z0-9 _\t;:,.\[\](|^&+=%!~<>{}?@\\\/#$-]* {
      if (yytext[yyleng-1] == '\n') {
                  lineno++;
            }
}

<IN_PHP>.+"/*".*"*/".*\n {
      if (yytext[yyleng-1] == '\n') {
                  lineno++;
            }
      //    printf("Commento eliminato\n");
}

<IN_PHP>.*"/*".*"*/".+\n {
      if (yytext[yyleng-1] == '\n') {
                  lineno++;
            }
      //    printf("Commento eliminato\n");
}

<IN_PHP>.+"/*".*\n {
      if (yytext[yyleng-1] == '\n') {
                  lineno++;
            }
      BEGIN(COMMENT);
}

<IN_PHP>{WHITESPACE} {
      int i;
      for (i=0; i<yyleng; i++) {
            if (yytext[i] == '\n') {
                  lineno++;
            }
      }
      /* elimina gli spazi, le tabulazioni e i newline nel codice
contando le righe */
}



<SINGLE_QUOTE>([^'\\]|\\[^'\\])+ {
      fprintf(tk,"T_ENCAPSED_AND_WHITESPACE\n");
      return T_ENCAPSED_AND_WHITESPACE;
}
```

```
<DOUBLE_QUOTES>[`]+ {
      fprintf(tk,"T_ENCAPSED_AND_WHITESPACE\n");
      return T_ENCAPSED_AND_WHITESPACE;
}

<BACKQUOTE>["]+ {
      fprintf(tk,"T_ENCAPSED_AND_WHITESPACE\n");
      return T_ENCAPSED_AND_WHITESPACE;
}

<DOUBLE_QUOTES,BACKQUOTE>{ENCAPSED_TOKENS} {
      fprintf(tk,"%c\n",yytext[0]);
      return yytext[0];
}


<IN_PHP,INITIAL,DOUBLE_QUOTES,BACKQUOTE,SINGLE_QUOTE>. {
      int r;
      if(r!=lineno)count++;
      printf("\033[01;32mriga %d:\033[00m",lineno);
      printf("\033[01;33m ERRORE LESSICALE: carattere non
riconosciuto '%c' (ASCII=%d)\033[00m\n",yytext[0],yytext[0]);
      r=lineno;

}

%%

segnalazione(){
if(warn>0)printf("                  \033[01;33me %d
warning\033[00m\n\n",warn);
else printf("\n");
}
```

3. ST.h

```
struct symrec
{
char *name;
int numero_parametri;
struct symrec *next;
};

typedef struct symrec symrec;
symrec *sym_table = (symrec*)0;
symrec *putsym();
symrec *getsym();

symrec *
putsym(char *sym_name,int num)
{

symrec *ptr;
ptr = (symrec *)malloc(sizeof(symrec));

ptr->name = (char*)malloc(strlen(sym_name));

strcpy(ptr->name,sym_name);
ptr->numero_parametri = num;
ptr->next = (struct symrec *)sym_table;
sym_table = ptr;
}

symrec *
getsym(char *sym_name,int num_p,int flag)
{
symrec *ptr;
for(ptr=sym_table;ptr!=(symrec*)0;ptr=(symrec*)ptr->next)
if(strcmp(ptr->name,sym_name)==0)
{
     if(ptr->numero_parametri != num_p && flag==1)
          yyerror("\033[01;34mERRORE SEMANTICO: numero parametri
della chiamata a funzione non corretto\033[00m\n");


return ptr;
}
return 0;


}
```