



POLITECNICO DI BARI
FACOLTA' DI INGEGNERIA
CORSO DI LAUREA MAGISTRALE
IN INGEGNERIA INFORMATICA

PROGETTO DI
LINGUAGGI FORMALI E COMPILATORI

PHP2C

Docente

Prof. Giacomo Piscitelli

Bavaro Gianvito
Capurso Domenico
Donvito Marcello

Anno accademico 2010-2011

INDICE

PARTE 1: DESCRIZIONE

Introduzione

Introduzione	1
Linguaggio PHP	2
Linguaggio C	5

Compilatori e interpreti

Compilatori e interpreti	7
Differenze tra compilatori e interpreti	8
Schema di funzionamento di un compilatore	10
Fase di analisi o front end	10
Fase di sintesi o back end	11

Analizzatore lessicale

Analizzatore lessicale	13
Funzionamento di uno scanner	14
Realizzazione di un analizzatore lessicale	15
Struttura del programma generato	18

Analizzatore sintattico

Analizzatore sintattico	19
Tipi di parser	19
Realizzazione di un analizzatore sintattico	20
Struttura del programma generato	23

Analizzatore semantico

Analizzatore semantico	25
------------------------------	----

Symbol Table

Funzionamento	28
Tipi di realizzazione	28
La libreria <i>Uthash</i>	29

Gestione degli errori

Gestione degli errori	30
Errori lessicali	30
Errori sintattici	31
Errori semantici	31
Strategie di riparazione	31

PARTE 2: IMPLEMENTAZIONE**Implementazione**

Implementazione	33
Sottoinsieme del linguaggio PHP considerato	33
Type Checking	36

Implementazione dello scanner

Implementazione dello scanner	37
Sezione Prologo	37
Sezione Regole	40

Implementazione del parser

Implementazione del parser	48
Sezione Prologo e dichiarazioni	48
Sezione Regole della grammatica	54
Lista di statement tra parentesi graffe	57
Definizione di costanti	58
Istruzione di diramazione if	59
Stampa a video: echo	61
Espressioni	64
Assegnazione	65

Operazione di somma	67
Definizione di un array	69
Common_scalar	70
Sezione Epilogo	73
pulizia ()	73
yyerror ()	73
main ()	74

Implementazione della Symbol table

Implementazione della Symbol table	77
Gestione della Symbol table	81
print_elements ()	81
find_element ()	81
delete_elements ()	82
add_elements ()	82
type_checking ()	84
type_array_checking ()	85
check_element_gen_code ()	87
check_index ()	88
check_element ()	90
echo_check ()	93
isconstant ()	96

Generazione del codice C

Generazione del codice C	97
Funzioni di generazione del codice	98
apri_file ()	98
chiudi_file ()	98
chiudi_cancella_file ()	99
cancella_file ()	99
gen_header ()	100
gen_constant ()	100
gen_expression ()	101

gen_echo_expression ()	102
print_expression ()	102
gen_create_array ()	103
gen_assignment ()	104
gen_if ()	105
gen_else_if ()	105
gen_while ()	105
gen_switch ()	106
gen_echo ()	106
gen_tab ()	107

Alcune funzioni utili

Alcune funzioni utili	108
Funzioni	109
clear ()	109
put_testo ()	110
get_testo ()	110
countelements ()	111
isnumeric ()	112

Il traduttore *PHP2C*

Struttura del traduttore <i>PHP2C</i>	113
Esecuzione del traduttore <i>PHP2C</i>	115

<i>Bibliografia e sitografia</i>	118
--	-----

APPENDICE

<i>php_lexer.l</i>	119
<i>php_parser.y</i>	127
<i>symbol_table.h</i>	138
<i>gen_code.h</i>	148
<i>utility.h</i>	153

INTRODUZIONE

Il progetto in questione si pone come obiettivo la realizzazione di un traduttore che sia in grado di “trasformare” un determinato sottoinsieme del linguaggio di programmazione PHP nel corrispondente linguaggio C.

Un traduttore è un programma che effettua la traduzione automatica da un linguaggio ad un altro. Solitamente i linguaggi utilizzati nelle applicazioni informatiche sono linguaggi formali, dove per linguaggio formale si intende un insieme di stringhe di lunghezza finita costruite sopra un alfabeto finito, cioè sopra un insieme finito di oggetti tendenzialmente semplici che vengono chiamati caratteri, simboli o lettere.

In altre parole, data una frase in un certo linguaggio formale che rappresenta il linguaggio sorgente, il traduttore si occupa di costruire una nuova frase appartenente ad un altro linguaggio formale che rappresenta il linguaggio destinazione, anche detto linguaggio target o pozzo.



Naturalmente il linguaggio sorgente e il linguaggio target sono equivalenti, nel senso che producono gli stessi risultati operando sugli stessi dati.

Tipicamente, i traduttori si occupano della traduzione di un programma scritto in linguaggio ad alto livello in un programma scritto in linguaggio di macchina. Tuttavia ciò non è vincolante, ed è possibile pertanto consentire anche la traduzione da un linguaggio ad alto livello ad un altro linguaggio ad alto livello, come nel nostro caso dove si procede con la traduzione da PHP a C, entrambi linguaggi di alto livello.



PHP (acronimo ricorsivo di “PHP: Hypertext Preprocessor”, preprocessore di ipertesti; originariamente acronimo di “Personal Home Page”) 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. 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.

A metà degli anni Novanta il Web era ancora formato in gran parte da pagine statiche, cioè da documenti HTML il cui contenuto non poteva cambiare fino a quando qualcuno non interveniva manualmente a modificarlo. Con l’evoluzione di Internet, però, si cominciò a sentire l’esigenza di rendere dinamici i contenuti, cioè di far sì che la stessa pagina fosse in grado di proporre contenuti diversi, personalizzati in base alle preferenze degli utenti, oppure estratti da una base di dati (database) in continua evoluzione.

PHP nasce nel 1994, ad opera di Rasmus Lerdorf, come una serie di macro la cui funzione era quella di facilitare ai programmatori l’amministrazione delle homepage personali: da qui trae origine il suo nome, che allora significava appunto Personal Home Page. In seguito, queste macro furono riscritte ed ampliate fino a comprendere un pacchetto chiamato Form Interpreter (PHP/FI). Essendo un progetto di tipo open source (cioè “codice aperto”, quindi disponibile e modificabile da tutti), ben presto si formò una ricca comunità di sviluppatori che portò alla creazione di PHP 3: la versione del linguaggio che diede il via alla crescita esponenziale della sua popolarità. Tale popolarità era dovuta anche alla forte integrazione di PHP con il Web server Apache (il più diffuso in rete), e con il database MySQL. Tale combinazione di prodotti, integralmente ispirata alla filosofia del free software, diventò ben presto vincente in un mondo in continua evoluzione come quello di Internet.

Alla fine del 1998 erano circa 250.000 i server Web che supportavano PHP: un anno dopo superavano il milione. I 2 milioni furono toccati in aprile del 2000, e alla fine dello stesso anno erano addirittura 4.800.000. Il 2000 è stato sicuramente l'anno di maggiore crescita del PHP, coincisa anche con il rilascio della versione 4, con un nuovo motore (Zend) molto più veloce del precedente ed una lunga serie di nuove funzioni, fra cui quelle importantissime per la gestione delle sessioni. La crescita di PHP, nonostante sia rimasta bloccata fra luglio e ottobre del 2001, è poi proseguita toccando quota 7.300.000 server alla fine del 2001, per superare i 10 milioni alla fine del 2002, quando è stata rilasciata la versione 4.3.0. La continua evoluzione dei linguaggi di programmazione concorrenti e l'incremento notevole dell'utilizzo del linguaggio anche in applicazioni enterprise ha portato la Zend a sviluppare una nuova versione del motore per supportare una struttura ad oggetti molto più rigida e potente.

Nasce così PHP 5, che si propone come innovazione nell'ambito dello sviluppo web open source soprattutto grazie agli strumenti di supporto professionali forniti con la distribuzione standard ed al grande sforzo di Zend che, grazie alla partnership con IBM, sta cercando di spingere sul mercato soluzioni di supporto enterprise a questo ottimo linguaggio. Lo sviluppo di PHP procede comunque con due progetti paralleli che supportano ed evolvono sia la versione 4 che la versione 5. Questa scelta è stata fatta poichè tuttora sono pochi i fornitori di hosting che hanno deciso di fare il porting dei propri server alla nuova versione del linguaggio.

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.

PHP implementa soluzioni avanzate che permettono un controllo completo sulle operazioni che possono essere svolte dal nostro server web. L'accesso ai cookie ed alle sessioni è molto semplice ed intuitivo, avvenendo attraverso semplici variabili che possono essere accedute da qualunque posizione all'interno del codice.

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.

Il modulo per eseguire script PHP è ormai installato di default sui server di hosting, e la comunità di sviluppatori risolve molto velocemente i bug che si presentano agli utenti.

A partire dal 2011 PHP non ha supporto nativo per le stringhe Unicode o multibyte, il che rende più complicato lo sviluppo di applicazioni multilingua per paesi che accettano caratteri speciali all'interno delle loro parole. Tuttavia, il supporto Unicode è in fase di sviluppo per una futura versione di PHP e consentirà stringhe che siano in grado di contenere caratteri non ASCII.



Il C è tecnicamente un linguaggio di programmazione ad alto livello. Tuttavia, poiché esso mantiene evidenti relazioni semantiche con il linguaggio macchina e l'assembly, risulta molto meno astratto di linguaggi anche affini (appartenenti allo stesso paradigma di programmazione), come per esempio il Pascal. Per questo motivo, talvolta viene anche identificato con la locuzione (più ambigua) di linguaggio di medio livello, se non addirittura (in modo certamente improprio) come macro-assembly, o assembly portabile.

Il C è rinomato per la sua efficienza, e si è imposto come linguaggio di riferimento per la realizzazione di software di sistema su gran parte delle piattaforme hardware moderne. La standardizzazione del linguaggio (da parte dell'ANSI prima e dell'ISO poi) garantisce la portabilità dei programmi scritti in C (standard, spesso detto ANSI C) su qualsiasi piattaforma.

Oltre che per il software di sistema, il C è stato a lungo il linguaggio dominante in tutta una serie di altri domini applicativi caratterizzati da forte enfasi sull'efficienza. Esempi tipici sono le telecomunicazioni, il controllo di processi industriali e il software real-time. Oggi il predominio del C in questi contesti è in parte diminuito a seguito dell'avvento di competitor significativi, primo fra tutti il C++; tuttavia, il tempo in cui il C si potrà considerare obsoleto appare ancora molto lontano.

Il C ha, e continua ad avere, anche una notevole importanza didattica, in quanto rappresenta un linguaggio di riferimento sia per la sua importanza tecnica, sia per la crescente importanza di linguaggi che derivano dal C (per esempio C++, Java e C#).

Il C è un linguaggio di programmazione relativamente minimalista; la sua semantica utilizza un insieme ristretto di concetti relativamente semplici e vicini al funzionamento dell'hardware dei calcolatori; molte istruzioni C sono traducibili direttamente con una singola istruzione di linguaggio macchina (per esempio, gli operatori di autoincremento e autodecremento). Nel linguaggio un ruolo centrale viene svolto dal concetto di puntatore, che viene generalizzato fino a coincidere con l'indirizzamento indiretto, un modo di accedere alla memoria hardware caratteristico di tutte le

moderne CPU. Questo rende il C un linguaggio particolarmente efficiente. D'altra parte, rispetto al linguaggio assembly il C ha in più una struttura logica definita e leggibile, funzioni in stile Pascal e soprattutto il controllo sui tipi (in fase di compilazione), che manca completamente in assembly. Inoltre la grammatica e la sintassi del C sono molto libere e flessibili, permettendo di scrivere istruzioni complesse e potenti in poche righe di codice (ma anche istruzioni assolutamente criptiche e illeggibili). In definitiva, il successo del C fu decretato dall'essere un linguaggio creato da programmatori esperti, per essere usato da programmatori esperti.

Questa grande libertà, la complessità sintattica del linguaggio (che come abbiamo visto contiene poche istruzioni di base) e il ruolo centrale dei puntatori, che è necessario usare praticamente fin dai primi programmi, ne fanno viceversa un linguaggio ostico e sconsigliabile ai neofiti, che cadono quasi subito in una serie di trappole che, se pure ovvie per un esperto, sono molto difficili da individuare per un principiante.

Grazie alla particolare efficienza del codice prodotto dai suoi compilatori, il C venne utilizzato per riscrivere la maggior parte del codice del sistema UNIX, riducendo l'uso dell'assembly ad un piccolo gruppo di funzioni. La sua importanza tuttavia, crebbe solo dopo il 1978 con la pubblicazione da parte di Brian Kernighan e Dennis Ritchie del libro *The C Programming Language* nel quale il linguaggio venne definito in modo preciso.

Il suo successivo larghissimo utilizzo portò alla nascita di diversi dialetti e quindi alla necessità di definirne uno standard: a questo scopo nell'estate del 1983 venne nominato un comitato con il compito di creare uno standard ANSI (American National Standards Institute) che definisse il linguaggio C una volta per tutte. Il processo di standardizzazione, il quale richiese sei anni (molto più del previsto), terminò nel dicembre del 1989, e le prime copie si resero disponibili agli inizi del 1990. Questa versione del C è normalmente chiamata C89. Lo standard venne anche adottato dall'International Organization for Standardization (ISO) nel 1999 con il nome di C Standard ANSI/ISO. Nel 1995 fu adottato l'Emendamento 1 al C Standard che, fra le altre cose, ha aggiunto nuove funzioni alla libreria standard del linguaggio. Usando come documento base il C89 con l'Emendamento 1, e unendovi l'uso delle classi di Simula, Bjarne Stroustrup iniziò a sviluppare il C++.

Il risultato finale del continuo sviluppo del C fu lo standard promulgato nel 1999, noto come ISO C99 (codice ISO 9899).

COMPILATORI E INTERPRETI

Solitamente i traduttori si suddividono in due grandi categorie: i *compilatori* e gli *interpreti*.

In realtà vi è anche una terza categoria rappresentata dai traduttori ibridi, come Java, che creano file con contenuto bytecode. Tale contenuto non è fruibile direttamente dal processore, ma deve essere interpretato dalla Java Virtual Machine (JVM), che in sostanza è un ulteriore strato di software che si interpone tra la macchina reale ed il programma. Questo rende quindi il programma indipendente dal processore.

Un'applicazione pratica è quella dei programmi per telefoni cellulari, condivisibili anche su apparecchi di marche o serie diverse, purché dotate di una JVM compatibile (cioè di una versione contenente le caratteristiche necessarie al funzionamento del programma).

I compilatori eseguono la traduzione sull'intero programma sorgente scritto in linguaggio ad alto livello. Quindi leggono il programma sorgente e lo traducono interamente nel programma oggetto. La traduzione viene realizzata tramite alcune fasi di analisi durante le quali si possono riscontrare degli errori, cioè delle non rispondenze alle regole formali del linguaggio. In caso di errori la traduzione non viene completata ed il programmatore viene informato sulla natura degli errori riscontrati e sulla loro posizione nel programma. Se non si riscontra alcun tipo di errore, il compilatore procede con la traduzione realizzando il programma scritto nel linguaggio target. Quindi il programma oggetto è generato solo se non ci sono errori sintattici, e la correttezza semantica è effettuata solo in fase di esecuzione.

Anche gli interpreti eseguono una fase di analisi e producono il programma nel linguaggio target. La sostanziale differenza rispetto ai compilatori è rappresentata dal fatto che l'interprete prende in esame un'istruzione alla volta, realizzandone la traduzione e l'esecuzione.

Quindi, per ogni istruzione del programma scritto in linguaggio sorgente, l'interprete analizza l'istruzione e verifica la presenza di errori, e in caso di errori, la traduzione viene arrestata. Nel caso in cui non ci sia alcun tipo di errore si procede con la produzione e l'esecuzione delle istruzioni nel linguaggio target corrispondente.

Pertanto l'interprete deve essere sempre attivo durante l'esecuzione del programma principale.

Anzi, per essere più precisi, si può affermare che l'unico programma in esecuzione è l'interprete, mentre il modulo sorgente costituisce i dati.

DIFFERENZE TRA COMPILATORI E INTERPRETI

I principali vantaggi forniti dagli interpreti sono rappresentati dalla possibilità di eseguire modifiche ed aggiunte al codice durante l'esecuzione, consentendo un debug interattivo e una migliore diagnostica. A seconda della struttura del linguaggio, le modifiche del programma possono richiedere reparsing o ripetizione dell'analisi semantica.

Inoltre gli interpreti supportano l'indipendenza dalla macchina.

Tuttavia, gli interpreti presentano anche numerosi svantaggi da tenere in considerazione. In particolare:

- Non è possibile effettuare ottimizzazione del codice;
- Durante l'esecuzione il testo del programma è continuamente riesaminato: i bindings, i tipi e le operazioni sono ricalcolate anche ad ogni uso. Per linguaggi molto dinamici questo rappresenta un overhead di 100:1 (o peggiore) nella velocità di esecuzione rispetto a quella del codice compilato. Per linguaggi più statici (C o Java), il degrado della velocità è dell'ordine di 10:1.
- Il tempo di startup per piccoli programmi è lungo, dato che deve essere caricato l'interprete e il programma deve essere parzialmente ricompilato prima dell'esecuzione.
- Si può verificare un sostanziale overhead di spazio: infatti l'interprete e tutte le routine di supporto di solito devono essere tenuti a disposizione.
- Il programma sorgente spesso non è compatto come se fosse compilato. Ciò può causare una limitazione nella dimensione dei programmi.

Di contro, i compilatori sono in grado di garantire l'ottimizzazione del codice, e risultano essere molto più veloci nell'esecuzione dei programmi. Tuttavia hanno una bassa facilità di messa a punto dei programmi.

Pertanto possiamo affermare che, dal punto di vista dell'efficienza, l'esecuzione di un programma compilato è molto più veloce di quella di un programma interpretato. Tuttavia l'uso di un interprete potrebbe essere di aiuto nelle prime fasi di sviluppo di un programma, in quanto permette un'immediata verifica delle funzionalità del codice realizzato.

Naturalmente, molti linguaggi (tra cui, C, C++ e Java) hanno gli interpreti (per il debug e lo sviluppo del programma) e i compilatori (per la produzione dell'applicazione). Alcuni linguaggi presentano traduzioni ibride.

Per quanto riguarda il nostro progetto, è stato deciso di implementare un compilatore che fosse in grado di tradurre un sottoinsieme del linguaggio PHP in linguaggio C. La scelta del compilatore è dovuta principalmente al fatto che la nostra traduzione prevede prima la lettura dell'intero programma in linguaggio sorgente, e poi la sua traduzione in linguaggio target. Inoltre ci garantisce anche una maggiore velocità di esecuzione.

SCHEMA DI FUNZIONAMENTO DI UN COMPILATORE

Il compilatore prende in ingresso un programma, il codice sorgente, su cui esegue una serie di operazioni in modo da ottenere, in assenza di errori, il codice oggetto. In generale i compilatori sono in grado di riconoscere alcune classi di errori presenti nel programma, e in alcuni casi di suggerire in che modo correggerli.



I compilatori attuali suddividono le operazioni di compilazione in due fasi fondamentali: la fase di **analisi**, anche detta *front end* e la fase di **sintesi**, anche detta *back end*.

Nella fase di analisi, viene creata una rappresentazione intermedia del programma sorgente, ossia il compilatore traduce il linguaggio sorgente in un linguaggio intermedio (di solito interno al compilatore). Invece nella fase di sintesi viene creato, partendo dalla rappresentazione intermedia, il programma target equivalente, ossia avviene la generazione del codice oggetto.

Fase di **analisi** o *front end*

In questa fase viene creata una rappresentazione intermedia del programma sorgente. La fase di front end quindi dipende dal linguaggio sorgente ma è indipendente dalla macchina target.

Il front end può essere suddiviso a sua volta in più fasi:

- **Analisi lessicale:** Attraverso un analizzatore lessicale, spesso chiamato *scanner* o *lexer*, il compilatore divide il codice sorgente in tanti "pezzetti", spezzoni di codice, chiamati *token*.

I token sono gli elementi minimi (non ulteriormente divisibili) di un linguaggio, ad esempio parole chiave (*for*, *while*), nomi di variabili (*pippo*), operatori (+, -, «).

- **Analisi sintattica:** L'analisi sintattica prende in ingresso la sequenza di token generata nella fase precedente ed esegue il controllo sintattico. Il controllo sintattico è effettuato attraverso una grammatica, e rappresenta il procedimento di costruzione della derivazione di una frase rispetto ad una data grammatica. Il risultato di questa fase è un albero di sintassi.
- **Analisi semantica:** L'analisi semantica si occupa di controllare il significato delle istruzioni presenti nel codice in ingresso. Controlli tipici di questa fase sono il *type checking*, ovvero il controllo di tipo, controllare che gli identificatori siano stati dichiarati prima di essere usati, e così via. Come supporto a questa fase viene creata una tabella dei simboli (*symbol table*) che contiene informazioni su tutti gli elementi simbolici incontrati durante la compilazione quali *nome*, *tipo*, *scope* (se presente) etc. Il risultato di questa fase è l'albero sintattico astratto (AST).
- **Generazione del codice intermedio:** Dall'albero di sintassi viene poi generato il codice intermedio.

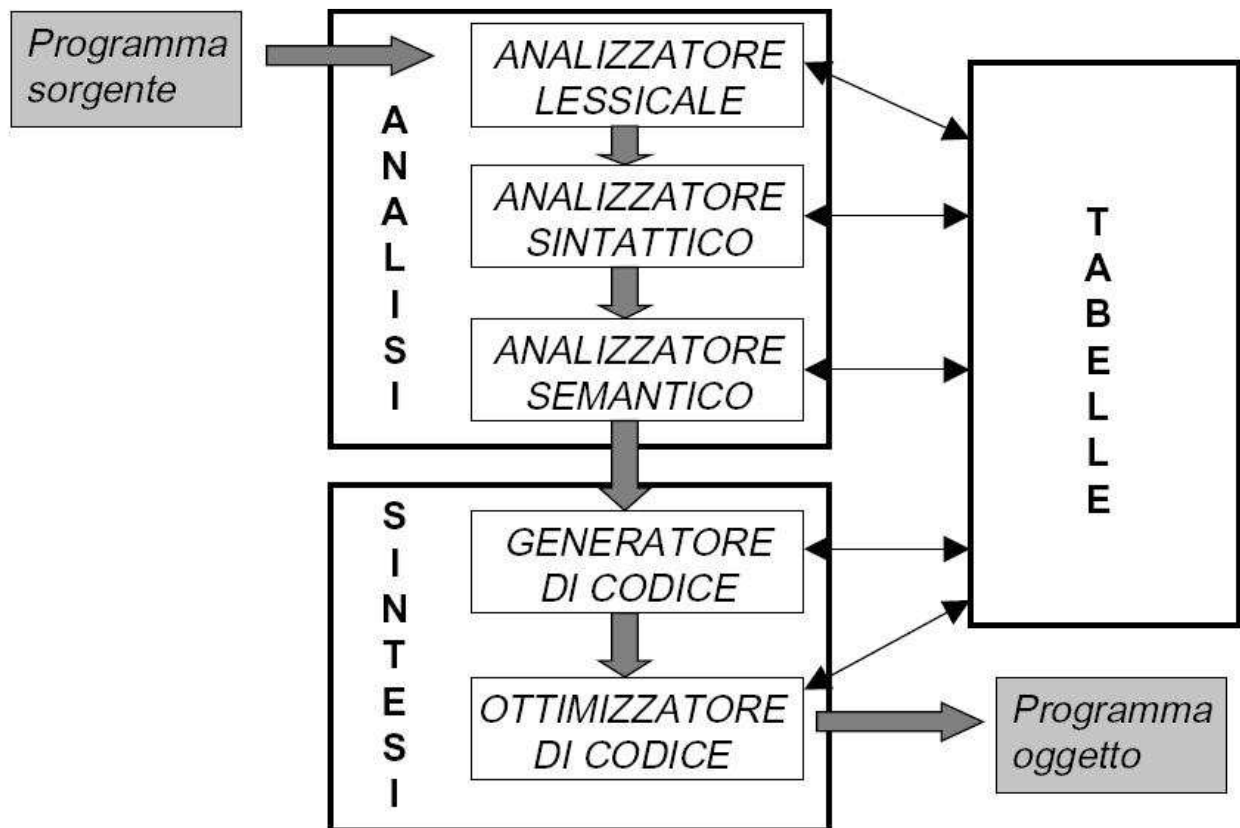
Fase di sintesi o back end

Nella fase di *back end* viene creato, partendo dalla rappresentazione intermedia, il programma target equivalente. La fase di back end è quindi indipendente dal linguaggio sorgente ma dipende dalla macchina target.

Anche questa fase può essere a sua volta suddivisa in più fasi:

- **Ottimizzazione del codice intermedio:** il codice viene analizzato e ottimizzato per una specifica architettura. L'ottimizzazione cerca di migliorare il codice per limitare il tempo di esecuzione e la memoria necessaria.
- **Generazione del codice target:** in questa fase viene generato il codice nella forma del linguaggio target. Spesso il linguaggio target è un linguaggio macchina.

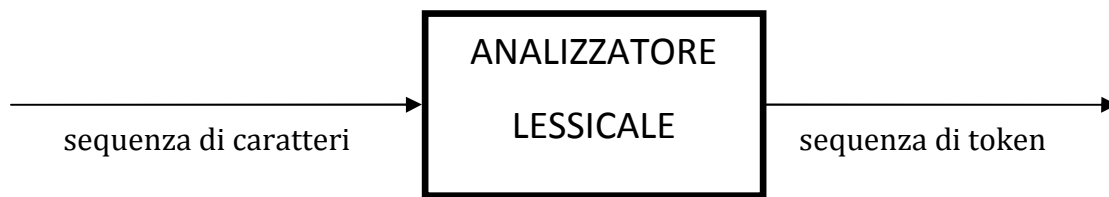
Di seguito viene mostrata un'immagine riassuntiva circa lo schema di funzionamento di un compilatore.



Nella fase di analisi rientra anche la generazione del codice intermedio, sul quale poi sarà applicata la fase di sintesi. Come è possibile notare dall'immagine, tutti i componenti impiegati durante l'esecuzione del compilatore hanno la possibilità di interagire con determinate tabelle. Queste tabelle, come ad esempio la symbol table, forniscono una serie di informazioni ausiliarie di grande importanza per la compilazione, di cui è possibile usufruire sia in fase di analisi che in fase di sintesi.

ANALIZZATORE LESSICALE

L'**analisi lessicale** consiste nel prendere in ingresso una sequenza di caratteri e produrre in uscita una sequenza di token. Il flusso di caratteri è genericamente il codice sorgente di un programma. L'analizzatore lessicale, a volte chiamato *scanner* o *lexer*, è lo strumento che si occupa dell'analisi lessicale.



Funzione di un analizzatore lessicale.

Un analizzatore lessicale legge il programma sorgente carattere per carattere e ritorna i token del programma sorgente. Quindi il compito principale di un analizzatore lessicale è quello di analizzare uno stream di caratteri in input e produrre in uscita uno stream di token classificando parole chiave, identificatori, operatori, costanti, ecc. In questa maniera lo scanner trasforma il codice sorgente in una forma compatta ed uniforme.

L'analizzatore lessicale è in grado di riconoscere e segnalare eventuali errori lessicali, eliminare eventuali informazioni non necessarie presenti nel codice sorgente (come i commenti e gli spazi bianchi), processare le direttive di compilazione (*include, define, etc.*).

Un *token* descrive un insieme di caratteri che hanno lo stesso significato. In particolare, il *token* o lessema è un elemento minimo (non ulteriormente divisibile) di un linguaggio che ha un tipo e un valore. I *token* costituiscono gli elementi base su cui andrà ad operare un analizzatore sintattico.

L'individuazione di token all'interno di uno stream di caratteri è effettuata attraverso pattern (schemi, modelli) descritti mediante espressioni regolari. L'analizzatore lessicale riconosce come token la stringa più lunga possibile.

Funzionamento di uno scanner

Per svolgere il loro compito, gli analizzatori lessicali si basano su un automa a stati finiti deterministico.

Un *Automa a Stati Finiti Deterministico* (DFSA) è costituito dalla quintupla

$$G = (X, E, \delta, x_0, X_m)$$

ove:

X è l'insieme finito degli stati;

E è l'insieme di simboli (alfabeto);

$\delta: X \times E \rightarrow X$ è la funzione di transizione di stato;

x_0 è lo stato iniziale;

X_m è l'insieme degli stati marcati o stati finali.

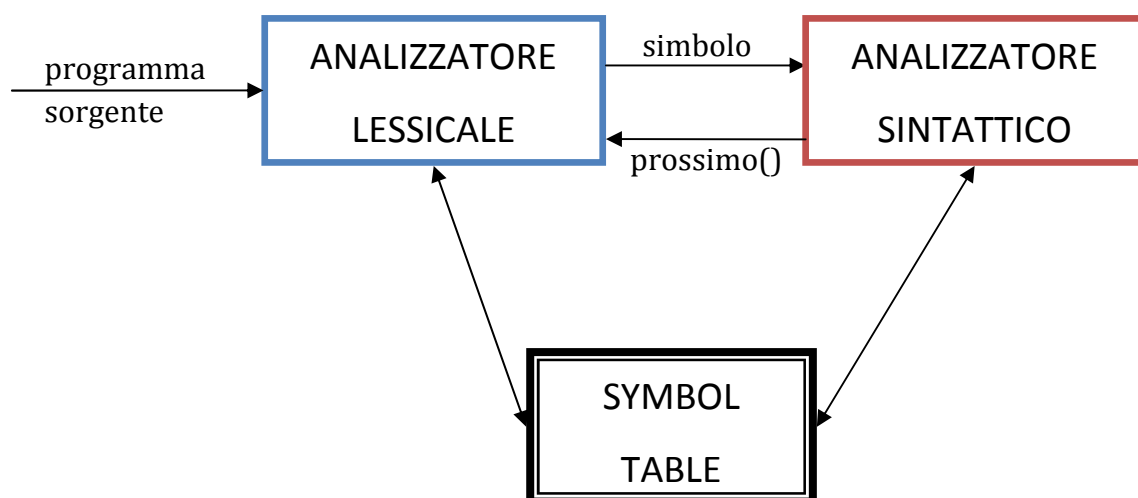
Ciascun elemento di X sta ad indicare uno stato del sistema e in genere descrive una particolare “*situazione*” o “*condizione operativa*” in cui può trovarsi lo scanner. In un automa finito deterministico si suppone che il numero di stati sia finito. La ricezione di un carattere da parte del *lexer* caratterizza un cambiamento di stato. La funzione δ stabilisce come avvengono le transizioni da uno stato all'altro. Ad esempio, se l'automa si trova nello stato $x \in X$ e si ha un cambiamento di stato in concomitanza con la ricezione del simbolo $e \in E$, il nuovo stato in cui viene a trovarsi il sistema è $\delta(x, e) \in X$.

Lo stato x_0 sta ad indicare lo stato di partenza del sistema (condizione iniziale), cioè lo stato dell'automa quando non è stato ricevuto alcun simbolo. Naturalmente, a seguito di uno o più transizioni di stato, l'automa può tornare in x_0 . Infine X_m è un sottoinsieme di X che raggruppa stati particolari che corrispondono a situazioni in cui il processo descritto ha completato compiti specifici. Per questo gli elementi di tale insieme sono anche detti *stati finali*. L'insieme X_m potrebbe essere composto anche da un unico stato e questo potrebbe anche coincidere con x_0 . Ciò corrisponderebbe a richiedere che il processo completi il suo compito riportandosi nella condizione di partenza. Quindi nell'esecuzione dell'automa, si parte dallo stato iniziale, e ci si sposta negli altri stati in base al carattere fornito in ingresso sino a quando non si raggiunge uno stato di accettazione nel quale si può inviare il token in output.

Realizzazione di un analizzatore lessicale

Un analizzatore lessicale può essere realizzato andando a scrivere a mano tutto il codice necessario per la sua implementazione. In tal caso si parla di realizzazione *hand-coded*, in quanto lo scanner viene codificato a mano. Tuttavia questo processo richiederebbe un costo molto elevato in termini di tempo, oltre che di complessità. Pertanto, sono stati realizzati degli strumenti di generazione automatica di *lexer*, che aiutano i programmatori nello sviluppo di uno *scanner*. Uno di questi tool è rappresentato da FLEX.

Il software FLEX (Fast LEXical analyzer generator), della Free Software Foundation, originariamente scritto in C da Vern Paxson nel 1987, è un free software generatore di scanner. Flex è pensato per la costruzione di analizzatori lessicali da interfacciare automaticamente con parser generati da Bison. Flex è comunque utilizzabile come generatore di programmi stand-alone.



Interazione tra FLEX e BISON.

Flex legge il file di input (o lo standard input) con la descrizione dello scanner da generare.

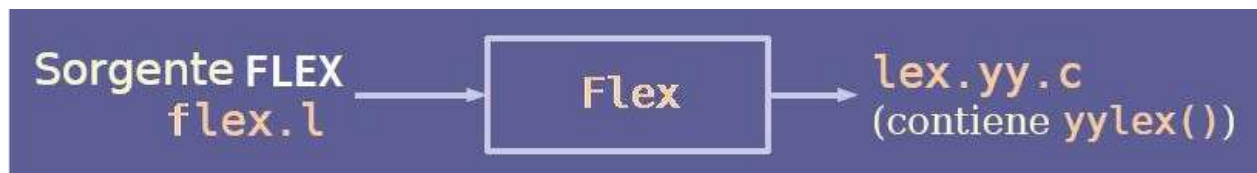
La descrizione in generale è fornita in forma di regole scritte come coppie di espressioni regolari e codice C. Questo file viene compilato e linkato assieme alla libreria *-lfl* per produrre un eseguibile. Flex è, quindi, un generatore che accetta in ingresso un insieme di espressioni regolari e di azioni associate a ciascuna espressione e produce in uscita un programma che riconosce tali espressioni.

Lo scopo di Flex è quello di partizionare l'input in un insieme di stringhe ciascuna in corrispondenza di una ben precisa regola sintattica. Al riconoscimento di una delle stringhe si può eseguire un corrispondente pezzo di codice fornito dall'utente.

I passi da seguire per avere un lexer a partire dal sorgente `file.l` sono:

```
flex file.l  
gcc -c lex.yy.c
```

L'output della compilazione di Flex è un programma C ed il nome è sempre `lex.yy.c`, come mostrato nell'immagine seguente.



Funzionamento di FLEX.

Il file sorgente per FLEX è composto di tre sezioni distinte separate dai simboli “%%”:

Prologo o dichiarazioni

%%

Regole o strutture sintattiche

%%

Epilogo o sezione routine ausiliarie

Parti di codice possono essere inserite sia nella prima parte (inserendole tra i simboli `%{` e `%}`) che nella seconda (tra `{ }` immediatamente dopo ogni espressione regolare che si vuole riconoscere), e vengono copiate integralmente nel file di output.

I commenti sono racchiusi, come in C, tra i simboli `/*` e `*/`.

La sezione delle regole è l'unica obbligatoria.

La **sezione prologo** comprende le *definizioni di base* cioè delle sottoespressioni regolari da utilizzare nella seconda sezione: in generale le espressioni regolari possono essere anche molto complesse, quindi le definizioni di base sono indispensabili per semplificare le espressioni regolari vere e proprie. Tale sezione può anche essere vuota e ogni riga della prima sezione il cui primo carattere non sia di spaziatura è una definizione.

Le definizioni consentono di dichiarare genericamente dei “name” secondo la forma

name definition

dove la definizione comincia al primo carattere diverso da blank che segue il nome e prosegue fino al termine della riga. Successivamente si potrà quindi far riferimento alla definizione mediante il suo nome. Le definizioni sono solitamente espresse mediante espressioni regolari.

Questa sezione può contenere, inoltre, le dichiarazioni in C per includere librerie e il file `parser.tab.h` prodotto da Bison, che contiene le definizioni dei token multi-caratteri, per definire variabili globali accessibili sia da `yylex()` che dalle routine e dai prototipi delle funzioni dichiarate nella sezione epilogo.

La seconda **sezione** contiene le **regole** sotto forma di coppie

espressione_regolare (pattern) {azione}

che vengono riconosciute dall’analizzatore lessicale.

Alcune utilizzano le definizioni della sezione prologo, racchiudendone il nome tra parentesi graffe. Ciascun pattern ha un’azione corrispondente, espressa mediante statement in codice C, che viene eseguito ogni volta che un’espressione regolare viene riconosciuta. I pattern devono essere non indentati e le azioni devono iniziare sulla stessa riga in cui termina l’espressione regolare e sono separate tramite spazi o tabulazioni. Se tale codice comprende più di una istruzione o occupa più di una linea deve essere racchiuso tra parentesi graffe. Se l’azione è nulla (azione espressa con il carattere “;”), quando viene incontrato il token esso viene semplicemente scartato.

Prima della prima regola si possono inserire istruzioni C (ad esempio commenti o dichiarazioni di variabili locali) ma solo se indentate o racchiuse tra parentesi graffe.

La **sezione epilogo** è opzionale e contiene le *procedure di supporto* di cui il programmatore intende servirsi per le azioni indicate nella seconda sezione: se è vuota, il separatore “%%” viene omissso. Tutte le righe presenti in questa sezione del programma sorgente sono ricopiate nel file `lex.yy.c` generato da Flex. Tale file è usato nelle routines che chiamano o sono chiamate dallo scanner.

Struttura del programma generato

Compilando il file `php_lexer.l` con il comando `flex php_lexer.l` si ottiene come risultato il file `lex.yy.c`, che contiene un programma C, il cui `main()` contiene la routine di scanning `yylex()` assieme ad altre routine ausiliari e macro.

La funzione `yylex()` viene richiamata ripetutamente dalla funzione principale del parser, `yyparse()`. Quando l'eseguibile dello scanner è in esecuzione, la funzione `yylex()` ad ogni invocazione analizza l'input file `yyin` alla ricerca di occorrenze delle espressioni regolari e, nel momento in cui ne trova una, ritorna il prossimo token ed esegue il codice C corrispondente (l'azione), per poi proseguire nella scansione dell'input. Se più di un pattern corrisponde all'input letto, Flex esegue l'azione associata all'espressione regolare che ha riconosciuto la sequenza più lunga o a quella dichiarata per prima in caso di pattern della stessa lunghezza, mentre se non viene trovata alcuna corrispondenza, si esegue la regola di default. Il successivo carattere di input viene considerato “matchato”, ricopiando sul file `yyout` il testo non riconosciuto carattere per carattere. Per default, i file `yyin` e `yyout` sono inizializzati rispettivamente a `stdin` e `stdout`. Se non specificato diversamente nelle azioni (tramite l'istruzione `return`), la funzione `yylex()` termina solo quando l'intero file di ingresso è stato analizzato. Al termine di ogni azione l'automa si ricolloca sullo stato iniziale, pronto a riconoscere nuovi simboli.

Flex mantiene il testo, letto dall'input e riconosciuto da una espressione regolare, in un buffer accessibile all'utente tramite le variabili globali `char *yytext` e `int yyleng`. Operando su tali variabili si possono definire azioni più complesse e passare al parser il testo letto dall'input.

Al termine dell'input (EOF), FLEX (ed in particolare la funzione `yylex()`) invoca la funzione `yywrap()` a cui fornisce le variabili globali `char *yytext` e `int yyleng`. Se `yywrap()` ritorna 0, si assume che `yyin` stia puntando ad un nuovo input file e la scansione continua attraverso la `yylex()`; se ritorna 1, lo scanner termina e a sua volta restituisce 0 al programma che l'ha chiamato.

ANALIZZATORE SINTATTICO

L'**analisi sintattica** è il processo atto ad analizzare uno stream continuo in input (letto per esempio da un file o una tastiera) in modo da determinare la sua struttura grammaticale grazie ad una data grammatica formale. Nella maggior parte dei linguaggi, tuttavia, l'analisi sintattica opera su una sequenza di token in cui l'analizzatore lessicale spezzetta l'input.

L'analizzatore sintattico, anche chiamato *parser*, è il programma che svolge questo compito.



Funzione di un analizzatore sintattico.

Tipi di parser

Il lavoro del parser è essenzialmente quello di determinare se e come l'input può essere derivato dall'assioma con le regole della grammatica formale. Questo può essere fatto essenzialmente in due modi:

- **Analisi top-down:** un parser può partire con l'assioma e cercare di trasformarlo nell'input. Intuitivamente, il parser parte dal più grande elemento e lo divide in parti sempre più piccole. I parser LL sono esempi di parser top-down.
- **Analisi bottom-up:** un parser può partire con l'input e cercare di riscriverlo sino all'assioma. Intuitivamente, il parser cerca di trovare il più elementare simbolo (simbolo non-terminale), quindi elabora gli elementi che lo contengono, e così via. I parser LR sono esempi di parser bottom-up.

Un'altra importante distinzione può essere effettuata in base al tipo di derivazione effettuata:

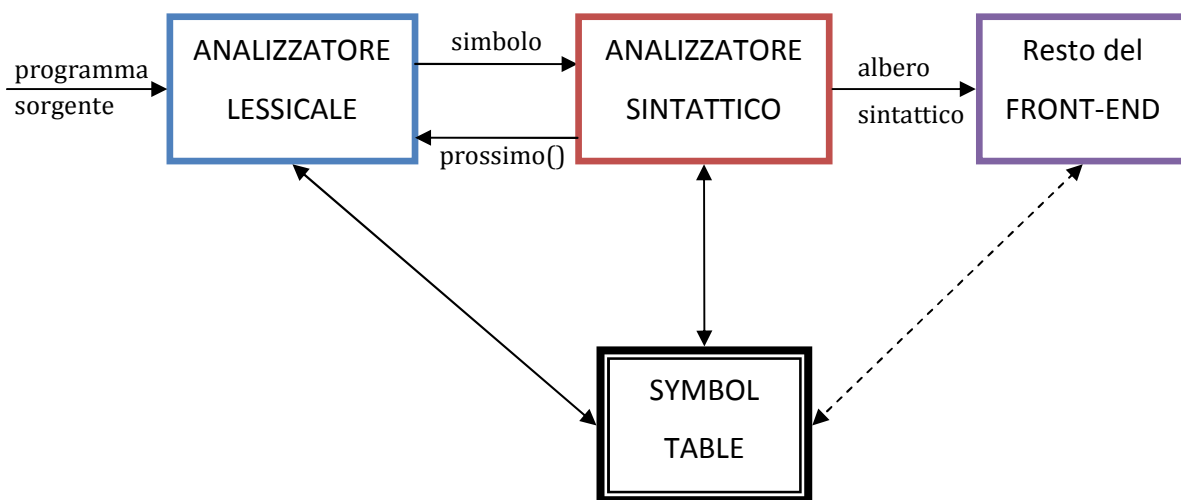
- **leftmost derivation:** derivazione nella quale durante ogni passo solo il non-terminale più a sinistra è sostituito. I parser LL generano una derivazione leftmost.

- **rightmost derivation:** derivazione nella quale durante ogni passo solo il non-terminale più a destra è sostituito. I parser LR generano una derivazione rightmost.

Realizzazione di un analizzatore sintattico

Così come per lo scanner, esistono degli strumenti automatici anche per la generazione del parser. Questi tool sono molto utili anche perché forniscono informazioni diagnostiche (ad esempio, in caso di grammatica ambigua, il tool riesce a determinare dove si crea l'ambiguità).

Il tool da noi utilizzato per generare il parser è BISON che esegue un'analisi di tipo bottom-up. Esso è un free software frequentemente usato con il generatore di lexer FLEX.



Interazione fra Bison e Flex.

Bison permette di descrivere le produzioni della grammatica del linguaggio da riconoscere e le azioni da intraprendere per ogni produzione.

Bison genera una funzione per riconoscere l'input e utilizza l'analizzatore lessicale per prelevare dall'input i token e riorganizzarli in base alle produzioni della grammatica utilizzata. Quando una produzione viene riconosciuta, viene eseguito il codice ad essa associato.

Ogni programma Bison consta di tre sezioni:

%{

Prologo

%}

Dichiarazioni

Regole della grammatica (o Strutture sintattiche)

%%

Epilogo (o Sezione routine ausiliarie)

La sezione delle regole è l'unica obbligatoria.

Tutti i caratteri di spaziatura (*blank*, *tab* e *newline*) presenti nel file sorgente del parser sono ignorati e i commenti sono racchiusi, come in C, tra i simboli */** e **/*.

Nella **sezione dichiarazioni** si definiscono alcune informazioni globali utili per interpretare la grammatica:

- *%token <simbolo>* : dichiara un simbolo terminale;
- *%left <simbolo>* : dichiara l'associatività a sinistra di un simbolo terminale;
- *%right <simbolo>* : dichiara l'associatività a destra di un simbolo terminale;
- *%nonassoc <simbolo>* : dichiara la non associatività di un simbolo terminale;
- *%type <type> simbolo* : dichiara un simbolo non terminale associandogli un tipo definito in *%union*;
- *%union* : dichiara l'intera collezione di possibili tipi di dati per i valori semantici. La keyword *%union* è seguita da una coppia di parentesi graffe che contengono la stessa cosa che va all'interno di una *union* in C;
- *%expect n* : dichiara il numero di conflitti di shift/riduzione. Normalmente Bison avverte se ci sono dei conflitti nella grammatica, ma la maggior parte delle grammatiche reali presentano dei conflitti di shift/riduzione che sono risolvibili in maniera prevedibile, ma difficili da eliminare. Pertanto è opportuno eliminare il warning di questi conflitti, a meno che il numero di conflitti cambi. Ciò viene fatto con la dichiarazione *%expect n*. *n* è un intero decimale. Quindi non ci saranno warning se ci sono *n* conflitti di shift/reduce e nessun conflitto reduce/reduce;

- **%start <simbolo>** : dichiara il simbolo iniziale della grammatica. Di default, si assume come simbolo iniziale il primo non terminale specificato dalla grammatica nella sezione seguente.

La **sezione regole** è il cuore del parser ed è composta da una o più produzioni espresse nella forma:

A: BODY {azione};

dove **A** rappresenta un simbolo non terminale e **BODY** rappresenta una sequenza di uno o più simboli sia terminali che non terminali.

I simboli **:** e **;** sono separatori. Nel caso la grammatica presenti più produzioni per lo stesso simbolo, queste possono essere scritte senza ripetere di volta in volta il non terminale, ma utilizzando il simbolo **"|"**.

Ad ogni regola può essere associata un'azione che verrà eseguita ogni volta che la regola viene riconosciuta. Le azioni sono istruzioni C e sono raggruppate in un blocco delimitato da parentesi graffe, possono apparire ovunque nel body di una regola e possono scambiare dei valori con il parser tramite delle pseudo-variabili introdotte dai simboli **\$\$**, **\$1**, **\$2**, ecc. dove la pseudo-variabile **\$\$** è associata al lato sinistro della produzione mentre le pseudo-variabili **\$n** sono associate all'n-esimo simbolo (terminale o non terminale) del corpo della produzione.

Bison ha un'azione di default che è **{ \$\$=\$1;}**: se si dovesse scrivere una produzione senza azioni semantiche, comunque il parser eseguirebbe la sua azione di default quando si tratterebbe di ridurre la produzione.

La **sezione epilogo** contiene tutte le routines C di supporto utili al corretto funzionamento del parser. In particolare, le tre routines più importanti che devono essere necessariamente presenti sono:

- **l'analizzatore lessicale yylex()**: Bison si aspetta un analizzatore lessicale di nome **yylex()** che ritorni al parser le tipologie di token dichiarati nella sezione relativa alle dichiarazioni e ne comunichi il valore (il lessema) tramite la variabile **yyval**. Se si usa un programma separato (Flex) per generare l'analizzatore lessicale si dovrà inserire **int yylex(void);** nella sezione "prologo" del programma **parser.y** per stabilire l'aggancio fra Bison e Flex;

- *la funzione principale `main()`*: avvia il procedimento di lettura, parsing e traduzione di un input. All'interno del file `parser.y` si potrebbe usare semplicemente:

```
main( )  
{  
    yyparse( );  
}
```

dove `yyparse()` è il parser generato da Bison che invoca ripetutamente `yylex()` innescando l'interazione parser-analizzatore lessicale;

- *la funzione di gestione errori `yyerror()`*: serve a gestire eventuali errori sintattici.

Struttura del programma generato

Compilando il file `parser.y` si ottiene il file `parser.tab.c` in cui viene definita la funzione `yyparse()`. Tale funzione è l'implementazione del parser: `yyparse()` riceve i token, esegue le azioni e termina quando incontra la fine dell'input o un errore di sintassi irreversibile o un errore semantico. La funzione `yyparse()` ritornerà il valore 0 se il parsing è terminato con successo, 1 se il parsing è fallito, per input non valido, oppure 2 se il parsing è fallito per mancanza di memoria.

Bison è ottimizzato per trattare le grammatiche LARL(1): tali grammatiche sono un sottoinsieme delle grammatiche LR(1) e si differenziano da esse poiché consentono di ottenere una tabella di parsing "*action + goto*" di dimensioni notevolmente inferiori.

Tuttavia, Bison, invece di generare un'unica matrice per il parsing, genera più tabelle di parsing in formato vettoriale. Non tutte queste tabelle sono utilizzate direttamente nella routine di analisi: alcune vengono utilizzate per stampare informazioni di debug altre per la gestione degli errori.

Alcune delle principali tabelle di parsing utilizzate nel file `parser.tab.c` sono:

- **YYTRANSLATE**: mappa i token di input in simboli numerici; ad ogni token nella grammatica Bison assegna un token number. Solo ai simboli effettivamente presenti nella grammatica è stato assegnato un numero di simbolo valido (il 2 indica un simbolo indefinito). Ogni volta

che *yyparse()* richiede un token, viene chiamata *yylex()* e viene restituito e tradotto il token di input nel corrispondente symbol number.

```

/* YYTRANSLATE[YYLEX] -- Bison symbol number corresponding to YYLEX. */
static const yytype_uint8 yytranslate[] =
{
  0, 2, 2, 2, 2, 2, 2, 2, 2, 2,
  2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
  2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
  2, 2, 2, 30, 66, 2, 2, 29, 2, 6,
  64, 38, 27, 24, 3, 25, 26, 28, 2, 2,
  2, 2, 2, 2, 2, 2, 2, 15, 65,
  20, 7, 21, 14, 2, 2, 2, 2, 2, 2,
  2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
  2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
  2, 68, 2, 69, 2, 2, 2, 2, 2, 2,
  2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
  2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
  2, 2, 2, 62, 2, 63, 31, 2, 2, 2,
  2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
  2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
  2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
  2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
  2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
  2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
  2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
  2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
  2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
  2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
  2, 2, 2, 2, 2, 2, 1, 2, 4, 5,
  6, 8, 9, 10, 11, 12, 13, 16, 17, 18,
  19, 22, 23, 32, 33, 34, 35, 36, 37, 39,
  40, 41, 42, 43, 44, 45, 46, 47, 48, 49,
  50, 51, 52, 53, 54, 55, 56, 57, 58, 59,
  60, 61
};

```

Tabella YYTRANSLATE contenuta nel file *parser.tab.c*.

- **YYDEFAC**: contiene le riduzioni da applicare in ogni stato cioè $yydefact[i]$ = numero della regola da usare nello stato i -esimo.

```

/* YYDEFECT[STATE-NAME] -- Default rule to reduce with in state
STATE-NUM when YYTABLE doesn't specify something else to do. Zero
means the default is an error. */
static const yytype_uint8 yydefect[] =
{
    0,      2,      0,      6,      1,      11,      5,      7,      0,      0,
    94,     90,      0,     147,     148,     150,     161,     157,     149,      0,
    24,      0,      0,      0,      0,      0,     145,      0,      4,      9,
    141,     43,     174,     174,     44,     12,     158,     151,     144,      0,
    156,      0,     159,     162,      0,      0,      0,      0,      0,      0,
      0,      0,     63,     64,     174,      0,     65,     159,     162,     11,
      0,      0,      0,      0,      0,      38,     39,      0,      0,     11,
      0,      0,      0,     104,     107,      0,     98,     101,     113,     110,
    116,     122,     125,     119,     128,     131,     134,     137,     140,     42,
      0,      0,      0,      0,      0,      0,      0,      0,     70,     86,
     83,     80,     77,     74,     72,     95,     91,      0,      0,     11,
      0,      0,     68,     41,      0,     40,      0,      0,      0,     11,
      0,      0,      0,      0,     165,      0,     13,      8,     10,      0,
    171,     175,     178,     172,     173,     152,     170,     153,      0,      0,
      0,      0,      0,      0,      0,      0,      0,      0,      0,      0,
      0,      0,      0,      0,     69,     85,     82,     79,     76,     73,
     89,     93,      0,      0,      0,      0,      0,      0,     11,     16,
     56,     163,     164,     62,     25,     11,     20,     55,     23,      0,
      0,     67,     35,     47,     37,     169,      0,     154,      0,     142,
      0,     103,     106,      0,     97,     100,     112,     109,     115,     121,
    124,     118,     127,     130,     133,     136,     139,     71,     87,     84,
     81,     78,     75,     56,     11,     59,      0,     22,     11,      0,
      0,      0,      0,     146,     167,     166,      0,      0,      0,     143,
     59,     56,      0,     60,     19,      0,     21,      0,     29,     36,
     48,     51,     46,      0,     14,     177,     176,     18,     59,      0,
     11,      0,      0,      0,      0,      0,     168,      0,     17,      0,
     61,     26,     11,     30,     49,     53,     54,      9,     15,     57,
      0,     45,     34,      0,      0,     11,     11,     27,     31,      9,
     58,      0,     11,     32,     11,     33
};

```

Tabella YYDEFAC contenuta nel file *parser.tab.c*.

ANALIZZATORE SEMANTICO

Nei compilatori la fase immediatamente successiva al parsing è rappresentata dall'**analisi semantica**.

Mentre il parser verifica la correttezza sintattica del programma sorgente, l'analizzatore semantico controlla se ci sono errori semantici nel programma sorgente e acquisisce l'informazione sui tipi che verrà utilizzata nella fase successiva di generazione del codice. Sostanzialmente l'analisi semantica effettua tutta una serie di controlli per verificare che ogni costrutto del codice sorgente venga usato consistentemente nel contesto in cui si trova, in particolare verifica la compatibilità di tipo (o *type checking*) di espressioni legate da operatori e la correttezza delle dichiarazioni degli identificatori. Una vera analisi semantica permetterebbe di individuare il significato del programma, cioè il suo scopo. Tali verifiche vengono effettuate sulla forma intermedia generata in precedenza.

Data la complessità della definizione della semantica di un linguaggio, la fase di analisi semantica non può essere automatizzata, come avviene invece per le fasi di analisi lessicale e analisi sintattica. Pertanto, il modulo che realizza la fase di analisi semantica deve essere realizzato manualmente.

Occorre precisare innanzitutto che l'analisi semantica dipende dalla semantica del linguaggio in input al compilatore. La semantica di un linguaggio è costituita da un insieme di regole che associano un significato ai costituenti lessicali e sintattici del linguaggio, cioè associano un significato ad una frase del linguaggio.

Definire la semantica di un linguaggio significa definire una funzione f che associa ogni frase del linguaggio L ad elementi di un dominio di interpretazione I :

$$f: L \rightarrow I$$

Il significato dell'intera frase viene quindi ottenuto con un processo compositivo, unendo il significato dei suoi costituenti.

Le regole semantiche possono essere espresse:

- *a parole* → dando luogo ad una semantica poco precisa e ambigua;
- *mediante azioni* → dando luogo ad una semantica operativa;
- *mediante funzioni matematiche* → dando luogo ad una semantica denotativa;
- *mediante formule logiche* → dando luogo ad una semantica assiomatica.

Il particolare tipo di regole semantiche utilizzate nell'ambito dei compilatori, senza dubbio il più diffuso nel campo dei linguaggi di programmazione, è quello operativo ed è noto sotto il nome di *grammatica ad attributi*.

Le grammatiche ad attributi estendono le grammatiche non contestuali introducendo le nozioni di:

- attributi associati ai simboli (terminali e non terminali) della grammatica;
- azioni semantiche e regole semantiche, che affiancano le regole sintattiche della grammatica.

Nelle grammatiche ad attributi:

- un *attributo* è una proprietà associata al simbolo, che può essere letta o assegnata dalle azioni semantiche (proprio come una variabile di un linguaggio di programmazione). Ogni simbolo terminale o non terminale può avere uno o più attributi. Gli attributi sono usati per rappresentare un "significato" (tramite un valore numerico, una stringa, una struttura di dati, ecc.) che viene associato ai simboli sintattici;
- ad ogni regola di produzione (*regola sintattica*) può essere associata una regola semantica, costituita da una sequenza di azioni semantiche in grado di assegnare valori agli attributi e di avere altri effetti (proprio come una o più istruzioni di un linguaggio di programmazione).

Gli attributi si distinguono in:

- *attributi sintetizzati*: sono gli attributi il cui valore dipende solo dai valori degli attributi presenti nel sottoalbero del nodo dell'albero sintattico a cui sono associati;
- *attributi ereditati*: sono gli attributi il cui valore dipende solo dai valori degli attributi presenti nei nodi predecessori e nei nodi fratelli del nodo dell'albero sintattico a cui sono associati.

Gli attributi sintetizzati realizzano un *flusso informativo ascendente* nell'albero sintattico (dalle foglie verso la radice) mentre gli attributi ereditati realizzano un *flusso informativo discendente* (dalla radice verso le foglie) e *laterale* (da sinistra verso destra e viceversa) nell'albero sintattico.

Essendo questo progetto basato sul parser Bison bottom-up, a ciascun simbolo risultano associati **attributi sintetizzati** il cui valore dipende solo dagli attributi presenti nel sottoalbero del nodo dell'albero sintattico a cui sono associati. Le regole sono identificate da azioni semantiche che vengono eseguite quando la regola di produzione è stata riconosciuta durante l'analisi sintattica.

SYMBOL TABLE

Nessuna delle fasi del compilatore può funzionare senza un posto all'interno del quale sia possibile memorizzare quanto scoperto durante la fase di analisi del programma dato in input. Questo posto è rappresentato dalla **tabella dei simboli** o **symbol table**.

Una funzione essenziale di un compilatore è quella di memorizzare gli identificatori che vengono usati nel programma sorgente insieme ai relativi valori di diversi attributi. Questi attributi possono essere, ad esempio, lo spazio in byte allocato per l'identificatore, il suo tipo, il suo scope (la parte di programma sorgente in cui l'identificatore ha significato). Ancora, se l'identificatore è il nome di una procedura, altri tipi di attributi sono il numero e i tipi di parametri che essa richiede, la modalità di passaggio di questi parametri (per valore, per riferimento, per nome), il tipo del valore ritornato, se c'è.

Funzionamento

Quando un nuovo identificatore viene trovato nel codice sorgente durante la fase di analisi lessicale esso viene inserito nella tabella dei simboli dall'analizzatore lessicale. Le fasi successive inseriscono informazioni nella tabella dei simboli e/o le usano in vari modi. Ad esempio durante la fase di analisi semantica e di generazione del codice intermedio c'è bisogno di sapere il tipo degli identificatori per controllare che siano usati propriamente e per generare le corrette operazioni su di essi. La fase di generazione del codice inoltre aggiunge alla tabella dei simboli anche informazioni dettagliate sullo spazio di memoria allocato per ogni identificatore.

Tipi di realizzazione

La *symbol table* è una struttura dati che contiene un record per ogni identificatore e in cui i vari campi del record sono degli attributi. Questa struttura dati deve consentire di poter individuare in maniera efficiente il record di un determinato identificatore che si vuole cercare e di poter leggere e/o scrivere nuovi valori al suo interno.

Esistono diversi modi attraverso i quali è possibile implementare una symbol table:

- *Unordered list*: per un insieme molto limitato di simboli;
- *Ordered linear list*: a causa dell'ordinamento può essere complessa da gestire;
- *Binary search tree*: il tempo di accesso è dell'ordine di $O(\log N)$;
- *Hash table*: è la tecnica più comunemente utilizzata ed è molto efficiente, a condizione però che lo spazio in memoria sia sufficientemente superiore al numero di record.

In questo progetto sarà utilizzata una sola symbol table, implementata mediante delle *Hash Tables*, ossia strutture dati che associano ogni record presente in tabella ad una chiave univoca; sulla chiave primaria viene applicata una funzione di hashing $h(k)$ che determina idealmente la posizione univoca in cui memorizzare la nuova entry. Con questo approccio si ottiene il grande vantaggio di rendere praticamente istantaneo il tempo di accesso e ricerca di ogni record.

La libreria *Uthash*

Per la realizzazione della symbol table si è scelto di utilizzare una libreria denominata *Uthash*, costituita da una serie di procedure di gestione degli Hash¹. *Uthash*, scritta da Troy Hanson, supporta operazioni tempo-costanti di aggiunta, ricerca e rimozione su strutture dati C.

Qualsiasi struttura o record in cui si sia definito un unico membro chiave di qualsiasi tipo, può essere gestito come Hash Table semplicemente aggiungendo il membro *UT_hash_handle* alla struttura. L'API mette quindi a disposizione le varie procedure con cui effettuare le operazioni di inserimento, ricerca e cancellazione di un elemento nella symbol table.

¹ Suggerimento carpito dalla relazione "A2C " *An ADA-like parser/scanner and C translator* di Ruggiero Campese A.A. 2010/2011.

GESTIONE DEGLI ERRORI

Una parte importante del processo di compilazione, oltre alla traduzione vera e propria, è il riconoscimento e la segnalazione degli errori nel programma sorgente. Ogni fase può incontrare errori.

Le fasi di analisi sintattica e semantica sono quelle che devono fronteggiare la maggiore frazione di errori che un compilatore è in grado di rilevare mentre poiché lo scanner ha una visione locale del programma sorgente, pochi sono gli errori individuabili nella fase di analisi lessicale.

L'analizzatore lessicale può rilevare un errore quando i caratteri rimanenti dell'input non formano nessun token del linguaggio. Quando lo stream di token viola le regole strutturali del linguaggio (la sintassi), il parser rileva un errore. Durante l'analisi semantica possono essere riconosciuti i costrutti che, pur essendo sintatticamente corretti, implicano operazioni non consentite fra gli operandi in gioco. Si pensi ad esempio all'addizione fra una variabile di un tipo primitivo ed un array.

È possibile suddividere gli errori in tre categorie: lessicali, sintattici e semantici.

Errori lessicali

Gli errori lessicali derivano da cattiva ortografia di identificatori, parole chiave, operatori o stringhe erroneamente non delimitate da apici.

Supponiamo, ad esempio, di trovare il seguente spezzone di codice all'interno del nostro programma sorgente:

$$wile(a < g(x))$$

Se ad esempio, in un programma, si incontra la stringa *wile* per la prima volta, lo scanner non sa distinguere tra un errore di battitura o una keyword corrispondente al nome di una funzione non dichiarata.

Errori sintattici

Gli errori sintattici sono determinati da una serie di disattenzioni nella stesura del codice, quali ad esempio possono essere il mal posizionamento del simbolo “;”, oppure parentesi tonde o graffe non bilanciate, comparsa di token che non sono regolarmente preceduti da altri token (ad esempio un *case* prima di uno *switch*), ecc.

Errori semantici

Infine, gli errori semantici sono quegli errori che operano sul significato assunto dal codice scritto, che anche se sintatticamente corretto potrebbe non avere senso per il compilatore. Questi errori possono essere dovuti ad una non corrispondenza tra operatori e operandi oppure tra il numero di parametri o tra il tipo di parametri, ecc.

Alcuni degli principali errori semantici che si possono incontrare in un programma possono essere dovuti ad una serie di motivi, quali ad esempio un identificativo non dichiarato, un tipo non valido, estremi di un indice non validi, operando non valido, operatore non ammesso, nome del tipo non valido, indice non valido, numero di indici errato, espressione non valida, assegnazione errata, nome del sottoprogramma errato, numero di parametri errato, operando non intero o reale, operando non booleano, tipi non compatibili, tipi non uguali, operando non puntatore, argomento non valido, campo record non valido, operando non array, operando non record, parametro attuale di tipo errato, passaggio parametro in modo errato, ecc.

STRATEGIE DI RIPARAZIONE

Dopo aver rilevato un errore, esso deve in qualche modo essere trattato per fare in modo che la compilazione continui e sia possibile rilevare eventuali altri errori. Questo perché un compilatore che si fermi una volta che ha trovato il primo errore non è poi di così tanto aiuto come si potrebbe pensare.

Per quanto attiene l'error recovery, le strategie più frequentemente adoperate sono:

- **panic mode:** dopo aver rilevato l'errore il parser riprende l'analisi in corrispondenza di alcuni token selezionati, detti *token sincronizzanti* (es.: delimitatori *begin end ; }* che hanno un ruolo chiaro e non ambiguo nel codice sorgente), scartando alcuni caratteri. Sostanzialmente, quindi, dopo ogni errore si esegue una sincronizzazione.

Svantaggi: può essere scartato molto input;

Vantaggi: si tratta di un metodo universale di semplice implementazione. Inoltre, funziona bene nel caso di pochi errori attesi all'interno di una stessa istruzione ed, infine, evita di produrre un loop infinito;

- **phrase level:** una volta rilevato l'errore, il parser può apportare correzioni locali sul resto dell'input inserendo/modificando/cancellando alcuni terminali per poter riprendere l'analisi (es.: segnala lo scambio di “,” con “;” oppure se trova un “if” e si accorge che manca il “;” segnala l'inserimento del simbolo “;”).

Svantaggi: potrebbero verificarsi dei loop infiniti. Inoltre risulta abbastanza difficoltoso trattare situazioni in cui l'errore è avvenuto molto prima del punto di rivelazione;

Vantaggi: si tratta di una tecnica molto utilizzata, in grado di correggere ogni tipo di stringa;

- **error productions:** se si ha un'idea circa gli errori più comuni, è possibile estendere la grammatica usando produzioni che generano costrutti erronei. Se il parser utilizza una tale produzione si notifica l'errore corrispondente e lo si può gestire di conseguenza.

Vantaggi: si tratta di un metodo molto efficiente per la fase di diagnostica;

- **global correction:** si cerca di trovare la migliore correzione possibile alla derivazione errata (minimo costo di interventi per inserzioni/cancellazioni/modifiche): esistono algoritmi che scelgono una sequenza minima di cambi per ottenere una correzione dal costo minimo. *Svantaggi:* si tratta di un metodo d'interesse teorico ma poco usato in pratica, se non per attuare la strategia di “phrase level”, in quanto molto costoso in termini di spazio e di tempo.

IMPLEMENTAZIONE

Dopo aver esposto le fasi principali attraverso le quali si articola la realizzazione del progetto, vengono ora descritte le modalità mediante le quali è stato possibile implementare il traduttore.

SOTTOINSIEME DEL LINGUAGGIO PHP CONSIDERATO

Prima di poter far ciò, però, è opportuno stabilire quale porzione del linguaggio sorgente PHP è stata trattata. Infatti, il progetto prevede la traduzione non dell'intero linguaggio PHP, ma soltanto di una porzione di esso. Ad esempio non sono state trattate tutte le operazioni e le istruzioni orientate agli oggetti, così come non è stata trattata la gestione delle funzioni.

In particolare sono stati gestiti i seguenti aspetti del linguaggio:

- Gestione delle variabili;
- Gestione delle costanti;
- Gestione degli array;
- Istruzioni di assegnazione;
- Operazioni di somma, differenza, prodotto, divisione e resto tra operandi (+, -, *, /, %);
- Operazioni di somma, differenza, prodotto, divisione e resto tra operandi in forma compatta (+=, -=, *=, /=, %=);
- Espressioni condizionali di uguaglianza (==), non uguaglianza (!=), maggiore (>), maggiore o uguale (>=), minore (<), minore o uguale (<=), and (&& o AND), or (|| o OR);
- Stampa a video (*echo*);
- Istruzioni di diramazione (*if* e *switch*);
- Iterazioni cicliche (*while*, *do while* e *for*).

Inoltre, per determinate istruzioni, sono stati stabiliti determinati vincoli da rispettare per poter garantire una corretta compilazione del codice sorgente. In particolare:

- Per quanto riguarda le **costanti**, la loro definizione viene effettuata tramite l'istruzione `define("NOME_COSTANTE", valore);`. Per evitare che il lexer possa avere problemi nella

distinzione tra una label e una costante, la definizione delle costanti è stata vincolata in maniera tale che il nome di una costante debba necessariamente iniziare con un solo underscore `_` e debba essere costituito da lettere maiuscole. Eventualmente è anche possibile inserire all'interno del nome della costante numeri o altri underscore.

- Per quanto riguarda le variabili, queste possono assumere valori di tipo intero (*int*), reale (*float*), stringa (*char **) o boolean (*bool*). Tuttavia, nel caso in cui si voglia utilizzare una **variabile numerica** (di tipo *int* o *float*) **che assuma un valore negativo**, è necessario porre tale valore tra parentesi tonde, indipendentemente dall'ambito di utilizzo. In caso contrario, verrà restituito un messaggio di errore.

Ad esempio per definire la variabile `$numero` che assume valore negativo pari a -3, è necessario scriverla ponendo il valore tra parentesi tonde, ossia `$numero = (-3);`

Questo procedimento deve essere svolto ogni qual volta si utilizza un valore negativo. Ad esempio in una somma di numeri negativi occorrerà scrivere: `$somma = (-5) + (-3);`

- Per quanto riguarda la **riassegnazione di una variabile già esistente**, dopo aver definito una variabile con un determinato tipo, è possibile riassegnare quella stessa variabile con un nuovo valore, purché sia dello stesso tipo del valore precedente, ossia della variabile stessa. In caso contrario si verifica un errore.

Ad esempio, se considero l'assegnazione `$a = 5;` la variabile `$a` sarà di tipo intero (*int*). Nell'assegnare un nuovo valore alla variabile, occorre utilizzare sempre un valore di tipo intero (ad es.: `$a = 3;`), altrimenti sarà prodotto un errore.

- Quando si vuole stampare qualcosa a video, si utilizza la funzione PHP **`echo()`**. Questa funzione consente di visualizzare a schermo tutto ciò gli viene passato come contenuto: stringhe, interi, float, boolean, variabili, costanti.

Esistono diversi modi attraverso i quali è possibile implementare questa funzione, come ad esempio tramite i doppi apici `""`, i singoli apici `'`, l'indicazione della sola variabile, l'utilizzo delle parentesi tonde `()`, ecc. Nella realizzazione del compilatore è stata prevista esclusivamente la modalità che fa uso dei doppi apici `""`. Pertanto qualsiasi cosa si voglia visualizzare a video deve essere racchiusa tra doppi apici.

Ad esempio, per visualizzare la stringa `"Hello World!!!"` occorre scrivere il contenuto della stringa tra doppi apici: `echo "Hello World!!!";`

Lo stesso se si vuole visualizzare una variabile `echo "$nome_variabile";` o una combinazione stringa e variabile: `echo "messaggio $nome_variabile";`

- Per quanto riguarda la **realizzazione di un array**, PHP prevede diverse modalità di definizione. Nella realizzazione del compilatore, invece, è stata prevista esclusivamente la modalità `$var=array(elementi);`
- Non sono stati previsti gli **array associativi**, quindi tutti gli array trattati possono avere come indice esclusivamente un numero intero non negativo. Nel caso in cui venga utilizzato un indice di tipo diverso da quello intero, si verifica un errore. Se l'indice è negativo viene sollevato un warning.
- Inoltre, tutti **gli elementi dell'array devono essere omogenei** tra loro, ossia dello stesso tipo. Nel caso in cui vi sia anche un solo elemento di tipo diverso dal tipo di tutti gli altri elementi dell'array, viene mostrato un errore.

Negli array non è possibile utilizzare valori passati per riferimento tramite il simbolo `&`, ma solo per valore.

TYPE CHECKING

La procedura di *type checking* consiste nell'effettuare una serie di controlli sui tipi delle diverse variabili coinvolte all'interno di una determinata operazione.

Nella seguente tabella sono riportate le modalità attraverso le quali viene gestito il controllo sui tipi, indicando per ogni operazione, a seconda del tipo degli operandi, il risultato ottenuto. Inoltre, viene indicato se il tipo assegnato al risultato dell'espressione produce o meno un messaggio di warning, ossia un messaggio che informa l'utente circa la correttezza della relativa espressione nel linguaggio target C.

TIPI	OPERAZIONE	RISULTATO	WARNING
int	somma(+) differenza (-) prodotto (*) divisione (/) resto (%)	int	no
float		float	no
char		int	yes
bool		int	yes
int, float		float	no
int, char		int	yes
int, bool		int	yes
float, char		float	yes
float, bool		float	yes
char, bool		int	yes
int, float, char		float	yes
int, float, bool		float	yes
int, char, bool		int	yes
float, char, bool		float	yes
int	Vale per l'incremento (++) e il decremento (--)	int	no
float		float	no
char		char	yes
bool		bool	yes
int	minore (<) minore o uguale (<=) maggiore (>) maggiore o uguale (>=)		no
float			no
char			yes
bool			yes
int, float			no
int, char			yes
int, bool			yes
float, char			yes
float, bool			yes
char, bool			yes

IMPLEMENTAZIONE DELLO SCANNER

Come già detto in precedenza, l'analizzatore lessicale realizzato tramite lo scanner generator FLEX si basa su di un unico file sorgente con estensione *.l*. Questo file è stato chiamato *php_lexer.l* ed è stato ottenuto restringendo e modificando opportunamente il file *zend_language_scanner.l* liberamente visionabile in qualsiasi versione del PHP e di proprietà dell'azienda Zend, creatrice dell'interprete ufficiale del linguaggio PHP.

Il file sorgente per FLEX è composto di tre sezioni distinte separate dai simboli "%%":

Prologo o dichiarazioni

%%

Regole o strutture sintattiche

%%

Epilogo o sezione routine ausiliarie

All'interno del file *php_lexer.l* non è stata utilizzata la sezione epilogo, in quanto non stono state previste procedure di supporto per le azioni indicate nella sezione regole. Pertanto il file contiene esclusivamente le sezioni prologo e regole.

Sezione Prologo

All'interno della sezione prologo, sono state sviluppate principalmente due parti:

- la definizione degli stati;
- la definizione dei pattern da riconoscere.

La definizione degli stati è stata effettuata sulla base del fatto che lo scanner altro non è che un automa finito deterministico, che parte da uno stato iniziale che per convenzione è pari a zero (0).

All'interno del lexer il primo stato iniziale è rappresentato dallo stato *INITIAL*.

Tuttavia è possibile definire nuovi e diversi stati rispetto a quello di default per l'automa, e ciò può essere fatto mediante il comando **%x NOME_STATO**.

Lo stato appena definito viene chiamato stato esclusivo di partenza.

In questo modo, quando si utilizza la funzione `BEGIN(NOME_STATO)`, lo scanner Flex shifterà nello stato `NOME_STATO`, e da questo momento in poi lo scanner riconoscerà solo i patterns che inizieranno con `<NOME_STATO>` fino a quando un'altra istruzione `BEGIN` sarà eseguita, comportando un ulteriore shift di stato. I pattern permessi per ogni stato sono descritti nella sezione regole.

In funzione di ciò sono stati definiti i seguenti stati:

```
%x ST_IN_SCRIPTING
```

Stato che indica che ci si trova all'interno dello script PHP.

```
%x ST_DOUBLE_QUOTES
```

Stato che indica che ci si trova all'interno di una sezione di codice che inizia con i doppi apici `""`.

```
%x ST_SINGLE_QUOTE
```

Stato che indica che ci si trova all'interno di una sezione di codice che inizia con i singoli apici `'`.

```
%x ST_COMMENT
```

Stato che indica che ci si trova all'interno di un commento multi riga che inizia con il simbolo `/*`.

```
%x ST_DOC_COMMENT
```

Stato che indica che ci si trova all'interno di un commento multi riga che inizia con il simbolo `/**`.

```
%x ST_ONE_LINE_COMMENT
```

Stato che indica che ci si trova all'interno di un commento esteso su di una sola riga che inizia con il simbolo `//` o `#`.

```
%option stack
```

L'ultimo rigo `%option stack` non rappresenta la definizione di uno stato, ma indica il settaggio di un'impostazione del compilatore. In particolare abilita l'utilizzo di uno stack per le condizioni iniziali, ossia i vari stati definiti in precedenza vengono gestiti all'interno di uno stack.

La seconda parte della sezione Prologo riguarda la descrizione delle definizioni, attraverso le quali è possibile specificare delle abbreviazioni per i pattern da riconoscere.

Di seguito sono mostrate le varie definizioni utilizzate all'interno del file `php_lexer.l`:

```
LNUM [0-9]+
```

che definisce tutti i numeri interi non negativi.

```
NLNUM ( "-" ) [0-9]+ ( "-" )
```

che definisce tutti i numeri interi negativi, che devono essere inseriti tra parentesi.

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

che definisce tutti i numeri reali non negativi.

NDNUM $(\"(-\"([0-9]^*\backslash.[0-9]^+)(\"\"))|(\"(-\"[0-9]^+\backslash.[0-9]^*)(\"\"))$

che definisce tutti i numeri reali negativi, che devono essere inseriti tra parentesi.

CONST_LABEL $\"_\"[A-Z][A-Z0-9_]^*$

che definisce tutte le possibili costanti che possono essere identificate.

LABEL $[a-zA-Z_\backslash x7f-\backslash xff][a-zA-Z0-9_\backslash x7f-\backslash xff]^*$

che definisce tutte le possibili stringhe di caratteri che possono essere identificate.

WHITESPACE $[\backslash n\backslash r\backslash t]^+$

che definisce tutti gli spazi bianchi.

TABS_AND_SPACES $[\backslash t]^*$

che definisce gli spazi e le tabulazioni orizzontali.

TOKENS $[;:,.\backslash[\backslash]())^&+ -/*=%!~$<>?@]$

che definisce i vari caratteri speciali che possono essere utilizzati.

ENCAPSED_TOKENS $[\backslash[\backslash]\{\}\$]$

che definisce le parentesi quadre e graffe, e il simbolo di dollaro.

ESCAPED_AND_WHITESPACE $[\backslash n\backslash t\backslash r\backslash \#'\.:;,()^&+ -/*=%!~<>?@]^+$

che definisce altri tipi di caratteri speciali.

ANY_CHAR $(. | [\backslash n])$

che definisce un punto o un new line.

NEWLINE $(\"\\r\" | \"\\n\" | \"\\r\\n\")$

che definisce un new line, ossia un ritorno a capo.

Infine, anche in questo caso è stato utilizzato il comando *option* per impostare una determinata opzione di Flex:

```
%option yylineno
%option noyywrap
```

Con *%option yylineno*, Flex viene settato in maniera tale da generare uno scanner che memorizza l'attuale numero di riga letto dall'input all'interno della variabile globale *yylineno*.

Invece con *%option noyywrap*, Flex viene impostato in maniera tale da far sì che lo scanner non chiama la funzione *yywrap()* sul end-of-file, ma semplicemente assume che non ci sono altri file da scansionare.

Nella sezione Prologo sono state inserite anche delle porzioni di codice racchiuse tra `%{` e `%}`.

```
%{  
    #include <stdio.h>  
    #include "php_parser.tab.h"  
}%
```

Questa porzione di codice viene implementata per includere sia la libreria di default relativa allo standard input-output `<stdio.h>`, sia il file `php_parser.tab.h` che viene prodotto da Bison. Quest'ultimo `#include` serve proprio a consentire l'interazione del lexer prodotto da Flex con il parser prodotto da Bison.

Un'altra porzione di codice inserita nella sezione Prologo è la seguente:

```
%{  
    int lineno = 1;  
    int comment_start_line = 0;  
}%
```

In questo caso vengono definite le variabili numeriche di tipo intero `lineno` e `comment_start_line`. La prima serve a tener traccia del numero di righe che sono state lette sino a quel dato momento, mentre la seconda serve a tener traccia del numero di riga a partire dalla quale è stato aperto un commento.

Sezione Regole

Nella sezione Regole vengono definite le varie regole che sono riconosciute dall'analizzatore lessicale. In particolare si utilizza una struttura di questo tipo:

`<NOME_STATO>` espressione regolare o pattern {azione}

Ciò significa che quando ci si trova nello stato iniziale esclusivo `NOME_STATO`, se lo scanner rileva una sequenza di caratteri che coincidono con l'espressione regolare o il pattern descritto, viene eseguita l'azione stabilita all'interno delle parentesi graffe.

Di seguito sono mostrate alcune delle regole principali utilizzate all'interno del file *php_lexer.l*:

```
<INITIAL>"<?php"([ \t]|{NEWLINE}) {  
    if (yytext[yylen-1] == '\n') {  
        lineno++;  
    }  
    BEGIN(ST_IN_SCRIPTING);  
    return T_INIT;  
}
```

In questo caso, quando nello stato iniziale *INITIAL* e viene rilevata la sequenza di caratteri "*<?php*" seguita da un carattere di newline o uno spazio o una tabulazione orizzontale, si incrementa eventualmente il numero di riga se è stato rilevato un carattere newline di ritorno a capo, e lo scanner passa allo stato *ST_IN_SCRIPTING* tramite la funzione *BEGIN*. Infine viene restituito al parser il token *T_INIT* per segnalare l'inizio dello script php.

```
<ST_IN_SCRIPTING>"if" {  
    return T_IF;  
}
```

Lo scanner si trova nello stato *ST_IN_SCRIPTING* e viene rilevata la sequenza di caratteri "*if*". In tal caso viene ritornato al parser il token *T_IF* che segnala la presenza di un if.

Questo tipo di regola è utilizzata per la rilevazione di tutte le parole chiave previste dal linguaggio, quando lo scanner si trova all'interno dello stato iniziale esclusivo *ST_IN_SCRIPTING* che rappresenta lo script PHP. Ogni volta che viene rilevata una determinata sequenza di caratteri come "*if*", "*elseif*", "*else*", "*while*", "*do*", "*for*", ecc. viene restituito al parser il token relativo.

Lo stesso viene fatto per alcuni tipi di operazioni, come ad esempio l'operazione di autoincremento (*++*), autodecremento (*--*), la condizione di uguaglianza (*==*), di disuguaglianza (*!=*), ecc.

```
<ST_IN_SCRIPTING>"++" {  
    return T_INC;  
}
```

In tal caso, quando lo scanner rileva all'interno dello script PHP la sequenza di caratteri "*++*", restituisce al parser il token *T_INC* che indica proprio l'operazione di auto-incremento.

Invece, per rilevare la presenza di un numero intero, un numero reale o una stringa, vengono utilizzate le seguenti regole che sfruttano le definizioni stabilite nella sezione Prologo precedente.

```

<ST_IN_SCRIPTING>{LNUM} {
    yylval.id = ( char * )strdup( yytext );
    return T_LNUMBER;
}

```

Lo scanner si trova nello stato *ST_IN_SCRIPTING* e viene rilevata la sequenza di caratteri *LNUM* che identifica un numero intero non negativo. In tal caso l'analizzatore lessicale mette il contenuto di *LNUM* all'interno dell'array *yytext*. Dopodiché viene effettuata una copia di *yytext* e assegnato a *yylval.id* il puntatore a questo valore. Questo perché, se fosse stato assegnato a *yylval.id* un puntatore a *yytext*, si sarebbero potuti verificare degli errori in quanto *yytext* viene successivamente sovrascritto dall'analizzatore lessicale mentre *yylval* è usato dal parser.

yylval è una variabile C utilizzata dal parser e definita come *union*. In particolare tale variabile è stata definita nel parser all'interno della *union* utilizzando un solo membro chiamato *id* che punta a una stringa di caratteri.

Dopo aver fatto puntare *yylval.id* al valore di *LNUM*, viene semplicemente passato al parser il token *T_LNUMBER* che rappresenta un numero intero.

```

<ST_IN_SCRIPTING>{NLNUM} {
    char *s = ( char * )malloc( ( strlen( yytext ) - 1 ) * sizeof( char ) );
    int j=0;
    int i;
    for(i=1;i<strlen(yytext)-1;i++) {
        s[j]=yytext[i];
        j++;
    }
    s[strlen(s)]='\0';
    yylval.id = s;
    return T_LNUMBER;
}

```

Qui lo scanner si trova nello stato *ST_IN_SCRIPTING* e viene rilevata la sequenza di caratteri *NLNUM* che identifica un numero intero negativo. In tal caso l'analizzatore lessicale si comporta alla stessa maniera del caso precedente, ma prima di assegnare a *yylval.id* il puntatore al valore *NLNUM*, viene eseguito un piccolo ciclo allo scopo di eliminare le parentesi tonde che circondano il numero negativo.

Innanzitutto viene creata la stringa *s* di dimensione pari al numero di elementi che costituisce *NLNUM* escluse le parentesi tonde. Dopodiché viene eseguito un ciclo che riempie questa stringa *s* con i caratteri che rappresentano il numero negativo. Fatto ciò, la stringa viene chiusa con il simbolo `'\0'` e il suo contenuto viene puntato da *yylval.id*.

Infine viene semplicemente passato al parser il token `T_LNUMBER` che rappresenta un numero intero.

Come mostrato di seguito, lo stesso processo viene svolto nel caso di numeri reali non negativi *DNUM*

```
<ST_IN_SCRIPTING>{DNUM} {
    yynval.id = ( char * )strdup( yytext );
    return T_DNUMBER;
}
```

e numeri reali negativi *NDNUM*.

```
<ST_IN_SCRIPTING>{NDNUM} {
    char *s=( char * )malloc( ( strlen( yytext ) - 1 )*sizeof( char )
    );
    int j=0;
    int i;
    for( i = 1; i < strlen( yytext ) - 1; i++) {
        s[ j ] = yytext[ i ];
        j++;
    }
    s[ strlen( s ) ] = '\0';

    yynval.id = s;
    return T_DNUMBER;
}
```

Anche per quanto riguarda la rilevazione delle stringhe si ha un comportamento del tutto simile al precedente.

```
<ST_IN_SCRIPTING>([ " ] ( [ ^$ _ " \ \ ] | ( " \ \ " . ) ) * [ " ] ) {
    yynval.id = ( char * )strdup( yytext );
    return T_CONSTANT_ENCAPSED_STRING;
}
```


Quando nello stato *ST_IN_SCRIPTING* viene individuata una sequenza di caratteri che inizia e termina con i doppi apici, essa viene puntata da *yylval.id* e passata al parser come token *T_CONSTANT_ENCAPSED_STRING*. Della stringa non faranno parte le sottostringhe che iniziano con *\$* e *_*. Tale regola sarà molto utile nella funzione *echo* per discriminare le variabili o le costanti dalle stringhe.

```
<ST_IN_SCRIPTING>{LABEL} {
    yynval.id = ( char * )strdup( yytext );
    return T_STRING;
}
```

Quando nello stato *ST_IN_SCRIPTING*, invece, viene individuata una sequenza di caratteri *LABEL*, senza che sia racchiusa tra doppi apici, essa viene puntata da *yylval.id* e passata al parser come token *T_STRING*.

Per quanto concerne la rilevazione delle costanti, si utilizza la seguente regola:

```
<ST_IN_SCRIPTING,ST_DOUBLE_QUOTES>{CONST_LABEL} {
    yynval.id = ( char * )strdup( yytext );
    return T_CONSTANT;
}
```

Quando nello stato *ST_IN_SCRIPTING*, invece, viene individuata una sequenza di caratteri *CONST_LABEL*, senza che sia racchiusa tra doppi apici, essa viene puntata da *yylval.id* e passata al parser come token *T_CONSTANT*.

```
<ST_IN_SCRIPTING>([`]({CONST_LABEL})[`]) {
    char *s = ( char * )malloc( ( strlen( yytext ) - 1 ) * sizeof( char ) );
    int j=0;
    int i;
    for( i = 1; i < strlen( yytext ) - 1; i++) {
        s[ j ] = yytext[ i ];
        j++;
    }
    s[ strlen( s ) ] = '\0';
    yynval.id = s;
    return T_CONSTANT;
}
```

Quando nello stato *ST_IN_SCRIPTING* viene individuata una sequenza di caratteri *CONST_LABEL* racchiusa tra singoli apici, essa viene elaborata in maniera tale da togliere i singoli apici all'inizio e

alla fine della sequenza di caratteri, e poi puntata da *yylval.id*. Infine viene passato al parser il token *T_CONSTANT*.

Per quanto riguarda le variabili, in PHP le variabili sono caratterizzate dal fatto che devono iniziare con il simbolo *\$* seguito da una stringa. Inoltre il primo carattere dopo il *\$* non può essere un numero.

Di seguito è mostrato il codice utilizzato per il riconoscimento di una variabile.

```
<ST_IN_SCRIPTING,ST_DOUBLE_QUOTES>"${LABEL} {  
    yyval.id = ( char * )strdup( yytext + 1 );  
    return T_VARIABLE;  
}
```

Quando lo scanner si trova in uno degli stati *ST_IN_SCRIPTING* o *ST_DOUBLE_QUOTES*, e viene rilevata una stringa preceduta dal simbolo *\$*, la stringa viene puntata da *yylval.id* e al parser viene passato il token *T_VARIABLE*.

Invece, nel caso in cui il primo simbolo immediatamente successivo al *\$* è un numero² (un caso non consentito dal php), viene effettuata una correzione automatica da parte del lexer che elimina il numero e rende accettabile la variabile.

```
<ST_IN_SCRIPTING,ST_DOUBLE_QUOTES>"${[0-9]}{LABEL} {  
    printf("\n\033[01;34mRiga:  %i.  CORREZIONE LESSICALE:  e' stato  
    corretto il nome della variabile \"%s\".\033[00m\n\n", yylineno, (  
    char * )strdup( yytext + 1 ) );  
    yyval.id = ( char * )strdup( yytext + 2 );  
    return T_VARIABLE;  
}
```

Infatti come è possibile notare, *yylval.id* punta alla copia del contenuto di *yytext + 2*, ossia alla stringa priva del simbolo *\$* e del carattere numerico. Dopodiché viene passato al parser il token *T_VARIABLE*. Nel caso vi sia più di un numero dopo il simbolo *\$* sarà attuata la medesima correzione, saltando tutte le cifre presenti dopo il simbolo *\$*, *yylval.id* punterà alla copia del contenuto di *yytext + 1 + num*, ossia alla stringa priva del simbolo *\$* e delle cifre erroneamente introdotte.

² Suggerimento carpito (e migliorato) dalla relazione *Compilatore PHP* di Rinaldi Matteo Marco e Villani Giuseppe. A.A. 2010/2011.

```

<ST_IN_SCRIPTING,ST_DOUBLE_QUOTES>"$"[0-9][0-9]+{LABEL} {
    printf("\n\033[01;34mRiga:  %i.  CORREZIONE LESSICALE:  e' stato
corretto il nome della variabile  \"%s\".\033[00m\n\n", yylineno, (
char * )strdup( yytext + 1) );
    char *str = ( char * )strdup( yytext );
    int num = 0;

    while( *str ) {
        if( isdigit( *str ) )
            num++;
        str++;
    }
    yylval.id = ( char * )strdup( yytext + 1 + num );
    return T_VARIABLE;
}

```

Di seguito sono mostrate le regole che determinano i principali passaggi di stato.

```

<ST_IN_SCRIPTING>["] {
    BEGIN(ST_DOUBLE_QUOTES);
    return "\"";
}

```

Quando all'interno dello script vengono rilevati di doppi apici " , lo scanner passa allo stato *ST_DOUBLE_QUOTES* e ritorna il simbolo " .

```

<ST_IN_SCRIPTING>['] {
    BEGIN(ST_SINGLE_QUOTE);
    return "'";
}

```

Lo stesso avviene nel caso nel singolo apice ' , dove lo scanner passa nello stato *ST_SINGLE_QUOTE* e viene tornato il simbolo ' .

Una volta entrati in uno di questi stati, per passare nuovamente allo stato *ST_IN_SCRIPTING* relativo allo script PHP generale, è necessario rilevare nuovamente un doppio apice " o un singolo apice ' a seconda dei casi.

```

<ST_DOUBLE_QUOTES>["] {
    BEGIN(ST_IN_SCRIPTING);
    return "\"";
}

```

```
<ST_SINGLE_QUOTE>[ ` ] {  
    BEGIN( ST_IN_SCRIPTING );  
    return ` ` ;  
}
```

Infine, quando nello stato *ST_IN_SCRIPTING* viene rilevata la sequenza di caratteri “?”>” seguita eventualmente da un carattere di newline, si torna nello stato iniziale *INITIAL* e viene restituito al parser il token *T_FINAL* per segnalare la fine dello script php.

```
<ST_IN_SCRIPTING>( " ? > " ) { NEWLINE } ? {  
    BEGIN( INITIAL );  
    return T_FINAL ;  
}
```

IMPLEMENTAZIONE DEL PARSER

L'analizzatore sintattico è stato realizzato tramite il parser generator BISON sulla base del contenuto di un file sorgente con estensione `.y`. Questo file è stato chiamato `php_parser.y` ed è stato ottenuto restringendo e modificando opportunamente il file `zend_language_parser.y` liberamente visionabile in qualsiasi versione del PHP e di proprietà dell'azienda Zend, creatrice dell'interprete ufficiale del linguaggio PHP.

Il parser rappresenta la componente principale del compilatore, in quanto gestisce tutta la grammatica ed è continuamente in comunicazione con il lexer e con la symbol table.

Anche il file `php_parser.y` è suddiviso in tre sezioni distinte separate dai simboli “%%”:

Prologo e Dichiarazioni

%%

Regole della grammatica o Strutture sintattiche

%%

Epilogo o sezione routine ausiliarie

Sezione Prologo e Dichiarazioni

Prima di definire la sezione di dichiarazione vera e propria, è stata introdotta una porzione di codice C all'interno della quale sono state definite una serie di informazioni ausiliarie utili per lo sviluppo dell'analizzatore sintattico. Questa porzione di codice, racchiusa fra i simboli `%{` e `%}`, contiene definizioni e dichiarazioni di funzioni e variabili che sono utilizzate all'interno delle azioni presenti nelle regole della grammatica.

Innanzitutto sono state inserite le principali librerie che saranno utilizzate dal programma:

```
#include <stdlib.h>
```

`stdlib.h` è una libreria che prevede una serie di funzioni e costanti di utilità generale. In particolare riguarda la conversione tra tipi, la gestione della memoria, il controllo dei processi, la ricerca ed ordinamento, operazioni di matematica semplice.

```
#include <stdio.h>
```

stdio.h è una libreria standard del linguaggio C che contiene definizioni di macro, costanti e dichiarazioni di funzioni e tipi usati per le varie operazioni di input/output e di manipolazione dei file.

```
#include <string.h>
```

string.h è una libreria standard del linguaggio C che contiene definizioni di macro, costanti e dichiarazioni di funzioni e tipi usati non solo nella manipolazione delle stringhe ma anche nella manipolazione della memoria.

```
#include "symbol_table.h"
```

symbol_table.h è il file utilizzato per la realizzazione della symbol table. Dato che il parser è costantemente in stretta comunicazione con la symbol table, è necessario includere questo file all'interno dell'analizzatore sintattico, per poter consentire poi un'analisi sintattico-semantic.

```
#define YYDEBUG 1
```

Attiva l'opzione di debugging, in maniera tale da mostrare a video di volta in volta tutti i passi che vengono eseguiti dal parser.

Dopodiché sono stati definiti alcuni degli elementi principali che saranno utilizzati in fase di esecuzione da parte del parser:

```
element_ptr element;
```

Indica un elemento della symbol table.

```
void yyerror( char *s );
```

Definisce la funzione *yyerror* per la gestione degli errori passando come parametro una stringa che descrive il tipo di errore che si è verificato.

```
extern int yylineno;
```

In questo modo vengono definite le variabili globali esterne utilizzate dallo scanner. La variabile *yylineno* è esterna al parser Bison in quanto è dello scanner Flex.

Vengono poi definite tutte le variabili e i puntatori che verranno utilizzati all'interno del parser.

```
char *current_value;
```

Stringa che contiene il valore corrente della variabile analizzata.

```
char *current_type;
```

Stringa che stabilisce il tipo corrente della variabile analizzata.

```
char *index_element = NULL;
```

Stringa che contiene l'indice dell'elemento di un array.

```
int _error = 0;
```

Variabile che funge da contatore degli errori.

```
int _warning = 0;
```

Variabile che funge da contatore dei warning.

```
int dim = 0;
```

Variabile che contiene la dimensione di un array, ossia il numero di elementi che lo compongono.

```
FILE *f_ptr;
```

Puntatore al file di uscita contenente il codice C tradotto a partire dal codice PHP fornito in ingresso.

```
bool array, read;
```

Variabili booleane che servono ad indicare, rispettivamente, se l'elemento in questione è un array o meno (*array*), e se l'elemento che sta venendo trattato faccia parte di un'espressione, anche complessa, di assegnazione ad un variabile o ad un elemento di un array (appartiene a un'operazione di scrittura *read* = false), oppure se fa parte di un'espressione di controllo che, quindi, non è assegnata a nessun elemento (appartiene a un'operazione di lettura *read* = true).

All'interno di questa porzione di codice, viene anche definita la funzione *check*, che effettua un controllo sulla symbol table.

```
void check( char *nameToken, char *offset, int nr, bool read )
{
    element = check_element( nameToken, offset, yylineno, read );
    current_value = element->value;
}
```

Questa funzione prende in ingresso come parametri la stringa *nameToken*, la stringa *offset*, l'intero *nr* e il boolean *read*. La stringa *nameToken* contiene il nome del token che è stato letto dall'analizzatore lessicale e passato al parser. La stringa *offset* indica l'indice dell'elemento dell'array che si vuole controllare. Questo indice è importante per stabilire se esiste o meno

nell'array un elemento in corrispondenza di quel dato indice. L'intero *nr*, invece, indica semplicemente il numero di riga in corrispondenza del quale è stato individuato il token, in maniera tale da far sì che in caso di errore sia possibile individuare facilmente l'errore tramite l'indicazione del numero di riga. Infine il boolean *read* è la stessa variabile specificata in precedenza.

La funzione *check* non ritorna alcun valore e serve a controllare se il token in questione esiste o meno all'interno della symbol table. Per far ciò si avvale della funzione *check_element* che si trova all'interno del file *symbol_table.h*, di cui si parlerà in maggior dettaglio quando si tratterà questo file. Ad ogni modo, per il momento basti sapere che tale funzione effettua il controllo, e se trova l'elemento nella symbol table restituisce l'elemento trovato, altrimenti fornisce un errore indicando che quell'elemento non è stato precedentemente definito.

Dopo aver fatto ciò, se l'elemento è presente nella symbol table, viene aggiornato il contenuto della stringa *current_value* con il valore dell'elemento ritornato dalla funzione *check_element*.

Terminata questa sezione di codice racchiusa tra *%{* e *%}*, si è proceduto con la parte di dichiarazione vera e propria del parser. Questa parte contiene le dichiarazioni che definiscono i simboli terminali e non terminali, specificano le precedenze, e così via.

In dettaglio:

```
%token T_IF
%token <id> T_LNUMBER T_DNUMBER T_STRING T_VARIABLE T_CONSTANT
T_NUM_STRING T_ENCAPSED_AND_WHITESPACE T_CONSTANT_ENCAPSED_STRING
%token T_CHARACTER
%token T_ECHO
%token T_DO
%token T_WHILE
%token T_FOR
%token T_SWITCH
%token T_CASE
%token T_DEFAULT
%token T_BREAK
%token T_CONTINUE
%token T_ARRAY
%token T_DEFINE
%token T_WHITESPACE
%token T_INIT
%token T_FINAL
```

Con *%token* definisco i diversi simboli terminali che mi vengono restituiti dal lexer.

Per quanto riguarda le precedenze:

```
%left ``,`
%left T_LOGICAL_OR
%left T_LOGICAL_AND
%left `=' T_PLUS_EQUAL T_MINUS_EQUAL T_MUL_EQUAL T_DIV_EQUAL
T_CONCAT_EQUAL T_MOD_EQUAL
%left `?' `:'
%left T_BOOLEAN_OR
%left T_BOOLEAN_AND
%left `+' `-' `.'
%left `*' `/' `%`
%left T_ELSEIF
%left T_ELSE
%left T_ENDIF
%left `)'`
```

In questo modo sono state dichiarate le precedenze a sinistra, ossia l'associatività a sinistra di un simbolo terminale, mentre di seguito sono mostrate le precedenze a destra, ossia l'associatività a destra di un simbolo terminale.

```
%right T_PRINT
%right `!`
%right `~` T_INC T_DEC
```

Infine, la non associatività di un simbolo terminale è stata dichiarata per i seguenti simboli terminali.

```
%nonassoc <id> T_IS_EQUAL T_IS_NOT_EQUAL
%nonassoc `<` T_IS_SMALLER_OR_EQUAL `>` T_IS_GREATER_OR_EQUAL
```

Per quanto riguarda la trattazione dei tipi, ogni elemento è stato definito come stringa:

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

Ciò significa che è stato prevista soltanto la stringa *id* come tipo per gli elementi. In altre parole, ogni elemento terminale o non terminale di tipo *<id>* viene considerato come una stringa.

L'assegnazione di un tipo di dato particolare ai simboli non terminali è realizzata mediante la parola chiave `%type`.

```
%type <id> variable r_variable w_variable element_array encaps_var
common_scalar;
```

In questo modo definisco tutti i non terminali che dovranno assumere il tipo `<id>` definito in `%union`.

```
%expect 52
```

Indica il numero di conflitti shift/reduce previsti dalla grammatica. In questo modo Bison non mostrerà warning se ci sono 52 conflitti di shift/reduce e nessun conflitto reduce/reduce. Nello specifico è riportata una tabella contenente tutti i conflitti di shift/reduce riscontrati nella nostra grammatica.

Numero	Descrizione
4	Conflitti shift/reduce a causa dell'ambiguità pendente fra le regole del costrutto <i>if</i> . Risolti mediante <i>shift</i> .
2	Conflitti shift/reduce a causa dell'ambiguità pendente sui costrutti <i>elseif/else</i> . Risolti mediante <i>shift</i> .
6	Conflitti shift/reduce a causa delle assegnazioni, semplici o in forma compatta, di valori a elementi di un array. Risolti mediante <i>shift</i> .
1	Conflitto shift/reduce a causa dell'ambiguità pendente fra le due regole del costrutto <i>for</i> . Risolto mediante <i>shift</i> .
39	Conflitti shift/reduce a causa dell'ambiguità (introdotta con azioni semantiche) pendente fra tutte le espressioni avente i simboli ' <i>'</i> ' e ' <i>'</i> '. Risolti mediante <i>shift</i> .

Tabella dei conflitti

Bison è un tool per generare parser *shift-reduce*, nello specifico un automa a stati finiti, in grado di effettuare un'analisi di tipo *bottom-up*. L'automa usa uno stack nel quale vengono memorizzati i simboli terminali e non terminali della grammatica in attesa di essere ridotti.

Il nome *shift-reduce* deriva dal fatto che le due operazioni fondamentali eseguite dal parser sono:

- *shift*, lettura di un simbolo dallo stream di input e inserimento nello stack;
- *reduce*, sostituzione degli ultimi k simboli inseriti $X_1 \dots X_k$ con il simbolo non terminale A , ma solo se esiste la produzione $A \rightarrow X_1 \dots X_k$.

La difficoltà di progettazione del parser risiede nella capacità di riconoscere quando è corretto operare uno shift e quando è corretto operare una riduzione, sulla base della grammatica definita. Quindi, sono introdotti due tipi di conflitti:

- ✚ *Conflitto shift-reduce*, il parser non sa decidere se fare uno shift o una reduce;
- ✚ *Conflitto reduce-reduce*, il parser non sa quale riduzione fare.

I conflitti di maggiore gravità sono quelli *reduce-reduce* per cui, nella ridefinizione della grammatica ci siamo prefissati di evitarli.

```
%start start;
```

start rappresenta il simbolo non terminale iniziale della grammatica, ossia l'assioma della grammatica.

```
%error-verbose
```

Impostando questa direttiva Bison fornisce una stringa del messaggio di errore più dettagliata e specifica.

```
%locations
```

Questa direttiva serve a specificare meglio la posizione dell'errore.

Sezione Regole della grammatica

Questa sezione rappresenta la parte principale e più importante del parser. Infatti in questa sezione viene descritta la grammatica e le azioni da compiere in corrispondenza di ciascuna regola. Si noti che le istruzioni che regolano la corretta indentazione del codice target sono state omesse per non appesantire la trattazione. Esse saranno visionabili nel codice riportato in appendice.

Consideriamo, ad esempio, l'assioma *start*:

```
start:
    T_INIT { cancella_file( ); f_ptr = apri_file( ); gen_header( f_ptr
    ); } top_statement_list
    ;
```

Il Body della regola relativa all'assioma prevede il simbolo terminale *T_INIT* e il non terminale *top_statement_list*. Il simbolo terminale *T_INIT* è fornito dal lexer quando viene rilevato uno script PHP. Fatto ciò, il compilatore inizierà ad elaborare il contenuto dello script scendendo nell'albero sintattico attraverso il non terminale *top_statement_list*.

Nella realizzazione del progetto è stato deciso di effettuare la generazione del codice C, ossia la traduzione della porzione di codice PHP in codice C, ogni volta che viene ad essere rilevata una data un'istruzione. Pertanto, in questo caso, quando sta per iniziare la compilazione del file PHP bisogna allo stesso tempo iniziare la generazione del codice di uscita della traduzione.

Questa operazione viene effettuata tramite l'esecuzione delle azioni semantiche poste tra parentesi graffe. In particolare in questo caso vengono eseguite le funzioni *cancella_file()*, *apri_file()*, *gen_header(f_ptr)* che si trovano all'interno del file *gen_code.h*.

Il file *gen_code.h* contiene una parte delle funzioni utilizzate dal parser per generare il codice C di traduzione nel file di uscita, la restante parte è presente nello stesso file *php_parser.y* mediante le funzioni *fprintf*.

A partire dall'assioma si passa al non terminale *top_statement_list*.

```
top_statement_list:
    top_statement_list top_statement
    | /* empty */
    ;
```

Questa regola della grammatica prevede due produzioni: la prima fa riferimento ai non terminali *top_statement_list* e *top_statement*, mentre la seconda fa riferimento al fatto che il contenuto dell'intero script PHP sia vuoto.

La prima regola di produzione è una regola ricorsiva in quanto il non terminale *top_statement_list* può richiamare se stesso. L'albero sintattico può essere ulteriormente espanso mediante il non terminale *top_statement*.

```
top_statement:
    statement
    ;
```

Questa regola di produzione richiama il non terminale *statement*.

```
statement:
    unticked_statement
    ;
```

Quando si realizza questa regola, innanzitutto si esegue l'azione di stampare all'interno del file il carattere `\t`, in maniera tale che ogni volta che si verifica uno statement venga inserita una tabulazione orizzontale per garantire una certa gerarchia tra le istruzioni ed ottenere una maggiore leggibilità del codice tradotto. Dopodiché si sviluppa il non terminale *unticked_statement*.

```
unticked_statement:
    '(' inner_statement_list ')'
| T_DEFINE '(' T_CONSTANT ',' common_scalar ')' { clear( );
add_element( $3, "constant", current_type, $5, 0, yylineno );
gen_constant( f_ptr, $3, current_type, $5 ); } ';'
| T_IF '(' expr ')' { gen_if( f_ptr, Exp ); clear( ); } statement
elseif_list else_single
| T_IF '(' error ')' statement elseif_list else_single
    { yyerror( "ERRORE SINTATTICO: espressione nel costrutto IF non
    accettata" ); }
| T_IF expr ')' statement elseif_list else_single
    { yyerror( "ERRORE SINTATTICO: '(' mancante nel costrutto IF" ); }
| T_WHILE '(' expr ')' { gen_while( f_ptr, Exp ); clear( ); }
while_statement { fprintf( f_ptr, "\n" ); }
| T_WHILE '(' error ')' while_statement
    { yyerror( "ERRORE SINTATTICO: espressione nel costrutto WHILE non
    accettata" ); }
| T_WHILE expr ')' while_statement
    { yyerror( "ERRORE SINTATTICO: '(' mancante nel costrutto WHILE" );
    }
| T_DO { fprintf( f_ptr, "do {\n" ); } statement T_WHILE { fprintf(
f_ptr, "} while( " ); } '(' expr ')' { print_expression( f_ptr, Exp );
fprintf( f_ptr, " );\n" ); clear( ); } ';'
| T_FOR
    '(' { fprintf( f_ptr, "for( " ); }
    for_expr
    ';' { fprintf( f_ptr, "; " ); }
    for_expr { print_expression( f_ptr, Exp ); clear( ); }
    ';' { fprintf( f_ptr, "; " ); }
    for_expr { clear( ); }
    ')' { clear( ); fprintf( f_ptr, " ) {\n" ); }
    for_statement { fprintf( f_ptr, "}\n" ); clear( ); }
| T_FOR '(' error ';' error ';' error ')' for_statement
    { yyerror( "ERRORE SINTATTICO: un argomento del costrutto FOR non è
    corretto" ); }
| T_SWITCH '(' expr ')' { gen_switch( f_ptr, Exp ); clear( ); }
switch_case_list { fprintf( f_ptr, "}\n" ); }
| T_SWITCH expr ')' switch_case_list
    { yyerror( "ERRORE SINTATTICO: '(' mancante nel costrutto SWITCH"
    ); }
```

```

| T_BREAK ';' { fprintf( f_ptr, "break;\n" ); }
| T_CONTINUE ';' { fprintf( f_ptr, "continue;\n" ); }
| T_ECHO echo_expr_list ';' {
    gen_echo( f_ptr, Exp, Phrase ); clear( );
    //Stampa gli avvisi se notice è uguale a 5 ( avviso riservato
    proprio alla funzione echo ).
    if( notice == 5 ) {
        printf( "\033[01;33mRiga %i. %s\033[00m", yylineno, warn[
notice ] );
        _warning++;
        notice = -1;
    }
| T_ECHO error ';'
    { yyerror ( "ERRORE SINTATTICO: argomento della funzione ECHO
errato" ); }
| expr ';' {
    if( countelements( Exp ) == 1 )
        print_expression( f_ptr, Exp );

    clear( );
    fprintf( f_ptr, ";\n" );
    //Stampa gli avvisi se notice è diverso da -1 e da 5 ( 5 è un
    avviso riservato alla funzione echo ).
    if( notice != -1 && notice != 5 ) {
        printf( "\033[01;33mRiga %i. %s\033[00m", yylineno, warn[
notice ] );
        _warning++;
        notice = -1;
    }
}
| ';' /* empty statement */
| end
;

```

Il simbolo non termale *unticked_statement* sviluppa diverse regole di produzione. Di seguito verranno esaminate le più importanti.

LISTA DI STATEMENT TRA PARENTESI GRAFFE

```
`{' inner_statement_list `}'
```

In questo caso si fa riferimento ad una lista di statement posti tra parentesi graffe.

```

inner_statement_list:
    inner_statement_list inner_statement
    | /* empty */
;

```

A sua volta il simbolo non terminale *inner_statement_list* può produrre ricorsivamente se stesso e il non terminale *inner_statement* o può essere vuoto.

```
inner_statement:
    statement
;
```

A sua volta *inner_statement* si sviluppa nuovamente sul simbolo non terminale *statement*.

DEFINIZIONE DI COSTANTI

```
T_DEFINE '(' T_CONSTANT ',' common_scalar ')' { clear( ); add_element(
$3, "constant", current_type, $5, 0, yylineno ); gen_constant( f_ptr,
$3, current_type, $5 ); } ';' ;
```

Questa regola della grammatica fa riferimento alla definizione di una costante.

La sintassi PHP relativa alla definizione di una costante è ***define("NOME_COSTANTE", valore);*** e viene gestita proprio da questa regola grammaticale.

Infatti è possibile notare subito il terminale *T_DEFINE* che gli viene passato dal lexer e che rappresenta la keyword *define*. Dopodiché vi è una parentesi tonda aperta e a seguire il token *T_CONSTANT*. Come detto in precedenza questo token fornito dal lexer identifica il nome di una costante. Successivamente vi è il simbolo di virgola e il non terminale *common_scalar*. *common_scalar*, come vedremo più avanti, fa riferimento al tipo di valore che viene assunto, ossia se si tratta di un intero, di un reale, di una stringa o di un boolean. In questo modo si assume il valore della costante, ed infine la parentesi tonda chiusa e il punto e virgola vanno a concludere l'istruzione.

Prima del simbolo ; viene inserita tra parentesi graffe l'azione da compiere quando viene riconosciuta questa istruzione. In particolare si tratta di 3 funzioni:

- *clear()*: è una funzione che serve a "svuotare" tre liste concatenate utili nelle fasi di type checking e traduzione. Questa funzione si trova all'interno del file *utility.h* che contiene alcune funzioni di ausilio alla realizzazione del compilatore;
- *add_element(\$3, "constant", current_type, \$5, 0, yylineno)*: funzione attraverso la quale l'elemento, in questo caso la costante, viene inserito nella symbol table. Per questo motivo tale funzione è contenuta all'interno del file *symbol_table.h*. Presenta sei parametri, ossia il nome della costante (*\$3*), la stringa "constant" per indicare che si tratta di una costante, il tipo della costante (*current_type*), il valore della costante (*\$5*), un numero indicante la

dimensione dell'array, se si tratta di un array (*0 in questo caso poiché è una costante*), e il numero di riga in corrispondenza della quale viene eseguita l'istruzione (*yylineno*);

- *gen_constant(f_ptr, \$3, current_type, \$5)*: funzione di generazione del codice che stampa nel file di uscita la corrispondente traduzione in codice C dell'istruzione di definizione di una costante. Passa quattro parametri, ossia il puntatore al file (*f_ptr*), il nome della costante (*\$3*), il tipo della costante (*current_type*) e il valore della costante (*\$5*). Questa funzione si trova naturalmente all'interno del file *gen_code.h*.

ISTRUZIONE DI DIRAMAZIONE IF

```
T_IF '(' expr ')' { gen_if( f_ptr, Exp ); clear( ); } statement
elseif_list else_single
```

Questa regola della grammatica fa riferimento all'istruzione di diramazione *if*.

Come è possibile notare essa è costituita dal terminale *T_IF* seguito dal simbolo di parentesi tonda aperta, il non terminale *expr*, il simbolo di parentesi tonda chiusa, il non terminale già visto in precedenza *statement*, il non terminale *elseif_list* e *else_single*.

Il non terminale *expr* fa riferimento a tutte le possibili operazioni che possono essere contenute all'interno di un'espressione, e sarà esaminato il seguito. Dopo la parentesi tonda viene svolta l'azione di stampare nel file di uscita la sintassi C dell'*if*, e di svuotare le tre liste concatenate.

```
elseif_list:
    /* empty */ { fprintf( f_ptr, "\n\t" ); }
| elseif_list T_ELSEIF '(' expr ')' { gen_elseif( f_ptr, Exp );
clear(); } statement { fprintf( f_ptr, "\n\t" ); }
;
```

Il non terminale *elseif_list* può dar vita ad un elemento vuoto (e in tal caso si stampa semplicemente un carattere di newline e tabulazione orizzontale finalizzando un solo *if*), oppure a un'altra produzione che prevede un eventuale *elseif_list* ricorsivo, seguito dal token *T_ELSEIF*, dall'espressione racchiusa tra parentesi tonde e dallo *statement*. In altre parole non si fa altro che garantire la ricorsività e riconoscere la keyword *elseif*. Il resto della regola grammaticale segue gli stessi criteri della precedente regola dell'*if*. Anche in questo caso, come azioni, vengono eseguite le stesse funzioni viste finora e la funzione *gen_elseif(f_ptr,Exp)* che semplicemente genera il codice C del *elseif* nel file di uscita.


```

else_single:
    /* empty */
| T_ELSE { fprintf( f_ptr, " else {\n" ); } statement { fprintf(f_ptr,
"\t}\n" ); }
;

```

Nell'*else_single* si ha una situazione del tutto analoga con la differenza che il token in questione è *T_ELSE* e il valore da stampare nel file di uscita è pari a *else*.

Nella gestione degli *if*, bisogna naturalmente prevedere anche eventuali situazioni di errore che si potrebbero verificare. Ciò viene fatto tramite le seguenti produzioni:

```

| T_IF '(' error ')' statement elseif_list else_single
  { yyerror( "ERRORE SINTATTICO: espressione nel costrutto IF non
    accettata" ); }
| T_IF expr ')' statement elseif_list else_single
  { yyerror( "ERRORE SINTATTICO: '(' mancante nel costrutto IF" ); }

```

Nella prima produzione viene considerata la possibilità che si possa verificare un errore all'interno della condizione di diramazione dell'*if*. Infatti il non terminale *expr* viene sostituito dalla keyword *error*. *error* è una keyword interna di Bison attraverso la quale il parser generator rileva un errore sintattico nel momento in cui legge un token che non può soddisfare nessuna regola della grammatica. Quando viene rilevato un errore, il parser segnala l'errore chiamando una funzione di segnalazione degli errori denominata *yyerror*. Questa funzione riceve un solo argomento che rappresenta il messaggio che dovrà essere visualizzato a video quando si verifica l'errore.

Nel caso in cui si verifichi un errore sulla condizione dell'*if*, verrà visualizzato il messaggio *"ERRORE SINTATTICO: espressione nel costrutto IF non accettata"*.

Allo stesso modo, la seconda produzione gestisce l'errore sintattico che si verifica quando viene omessa la parentesi tonda aperta, visualizzando il messaggio di errore *"ERRORE SINTATTICO: '(' mancante nel costrutto IF"*.

Lo stesso tipo di meccanismo viene applicato nella gestione dei cicli *while*, *do while*, *for* e dei costrutti *switch*, *break* e *continue*.

STAMPA A VIDEO: ECHO

In PHP, per visualizzare a video un qualsiasi tipo di informazione, si utilizza la funzione ***echo()***.

Nel progetto viene trattata la forma sintattica ***echo "qualsiasi cosa"***; gestita tramite la seguente regola grammaticale.

```
T_ECHO echo_expr_list ';' {
    gen_echo( f_ptr, Exp, Phrase );
    clear( );
    if( notice == 5 ) {
        printf( "\033[01;33mRiga %i. %s\033[00m", yylineno, warn[
notice ] );
        _warning++;
        notice = -1;
    }
}
```

Essa prevede la presenza del token *T_ECHO* fornito dal lexer, seguito dal non terminale *echo_expr_list* e dal simbolo *;*. In corrispondenza di questa regola, viene eseguita la funzione *gen_echo(f_ptr, Exp)* che scrive il corrispondente codice C nel file di uscita, e la funzione *clear()* che svuota il contenuto di tre liste concatenate. Nel caso in cui la variabile *notice* assuma valore uguale a 5, viene visualizzato un messaggio di warning indirizzato esclusivamente alla funzione *echo*.

Il non terminale *echo_expr_list* è sviluppato dalla seguente regola grammaticale.

```
echo_expr_list:
    ``` encaps_list ```
 | T_CONSTANT { echo_check($1, 0, yylineno); }
 | T_CONSTANT_ENCAPSED_STRING { put_testo(&Exp, $1); }
 | r_variable { echo_check($1, index_element, yylineno); }
 ;
```

In particolare vengono considerate quattro produzioni:

1. il caso in cui si voglia visualizzare a video qualcosa posto tra doppi apici, solitamente un insieme di elementi di diverso tipo come variabili, elementi di un array o costanti;
2. il caso in cui si voglia visualizzare a video una costante definita dall'utente;
3. il caso in cui si voglia visualizzare a video una stringa posta tra doppi apici;
4. il caso in cui si voglia visualizzare a video una variabile senza doppi apici;

Consideriamo le singole produzioni.

Nel primo caso è prevista la presenza del non terminale *encaps\_list* tra doppi apici.

```
encaps_list:
 encaps_list encaps_var{ echo_check($2, index_element, yylineno); }
| encaps_list T_STRING { strcat($2, " "); put_testo(&Phrase, $2); }
| encaps_list T_NUM_STRING { put_testo(&Phrase, $2); }
| encaps_list T_ENCAPSED_AND_WHITESPACE { put_testo(&Phrase, $2); }
| /* empty */
;
```

Come è possibile notare dalla relativa regola, il non terminale *encaps\_list* si può espandere in maniera ricorsiva individuando una variabile (*encaps\_var*), una stringa non racchiusa tra apici (*T\_STRING*), un numero compreso fra doppi apici (*T\_NUM\_STRING*), alcuni caratteri speciali (*T\_ENCAPSED\_AND\_WHITESPACE*) o nessun simbolo.

Ogni qual volta trova uno di questi elementi, viene concatenato uno spazio vuoto tramite la funzione *strcat* e inserito per riferimento il contenuto dell'elemento all'interno di una lista concatenata di stringhe denominata *Phrase* tramite la funzione *put\_testo* che si trova nel file *utility.h*. Tale lista servirà per la traduzione di frasi libere poste nella funzione *echo* in C.

```
encaps_var:
 T_VARIABLE { $$=$1; index_element = 0; }
| T_VARIABLE '[' T_NUM_STRING ']' { $$=$1; index_element = $3; }
| T_VARIABLE '[' T_VARIABLE ']' { $$=$1; index_element = $3; }
| T_CONSTANT { $$=$1; index_element = 0; }
;
```

Considerando il non terminale *encaps\_var*, è possibile notare come faccia riferimento all'individuazione di una variabile. In particolare prevede quattro produzioni, a seconda che si tratti di una semplice variabile, di un elemento di un array il cui indice è definito tramite un numero compreso fra doppi apici, di un elemento di un array il cui indice è a sua volta una variabile, di una costante.

Questa distinzione è necessaria in quanto occorre verificare che la variabile, l'elemento dell'array (e l'indice, se è una variabile) o la costante che si vuole visualizzare esista, ossia sia già stata definito in precedenza.

Per effettuare questo controllo viene eseguita la funzione *echo\_check* che si trova nel file *symbol\_table.h*, in quando deve controllare se l'elemento esiste o meno nella symbol table.

La funzione *echo\_check* passa tre parametri: il nome della variabile o dell'array (*\$1*), l'indice dell'array se si tratta di un array (*index\_element*) e il numero di riga corrente (*yylineno*).

Nel secondo caso si vuole stampare una costante, ossia il token `T_CONSTANT`. Essa può essere stampata solo se è stata definita dall'utente e quindi se è presente nella Symbol table, per questo motivo è utilizzata la stessa funzione `echo_check`.

Nel terzo caso si vuole stampare a video una stringa posta tra doppi apici, ossia il token `T_CONSTANT_ENCAPSED_STRING`. Anche in questo caso si esegue la funzione `put_testo` per inserire il valore del token all'interno della struttura di testo *Exp*.

Nel quarto caso, si vuole stampare a video direttamente una singola variabile identificata dal non terminale `r_variable`. Essendo una variabile, occorre controllare che sia stata precedentemente definita, quindi che sia presente nella symbol table, tramite la stessa funzione `echo_check`.

È utilizzato il non terminale `r_variable` in quanto, essendo un *echo*, la variabile dovrà solo essere letta.

```
r_variable:
 variable { $$ = $1; read = true; }
;
variable:
 T_VARIABLE { $$ = $1; }
 | element_array
;
```

Infatti la regola grammaticale di `r_variable` non fa altro che considerare una variabile in sola lettura, quindi facente parte di una espressione generica o di assegnazione, impostando il flag `read` a `true`. La variabile, a sua volta, potrà essere una semplice variabile o l'elemento di un array.

Anche per quanto riguarda la gestione dell'*echo* è stata prevista la presenza di errori tramite la keyword `error` di Bison.

```
T_ECHO error ';'
{
 yyerror ("ERRORE SINTATTICO: argomento della funzione ECHO errato");
}
```

In caso di errore, tramite la funzione `yyerror` verrà visualizzato il messaggio *"ERRORE SINTATTICO: argomento della funzione ECHO errato"*.

**ESPRESSIONI**

Un'altra produzione molto importante del non terminale *unticked\_statement* è rappresentata dalla seguente regola grammaticale.

```

expr ';' {
 if(countelements(Exp) == 1)
 print_expression(f_ptr, Exp);
 clear();
 fprintf(f_ptr, ";\n");
 if(notice != -1 && notice != 5) {
 printf("\033[01;33mRiga %i: %s.\033[00m", yylineno, warn[notice]);
 notice = -1;
 }
}

```

Le espressioni sono moltissime e possono essere di diversi tipi. In ogni caso quando si verifica una produzione di questo tipo si procede con lo stampare nel file di output una variabile, un elemento di un array o una costante, se la lista ha un solo elemento, o un punto e virgola con ritorno a capo, e infine, nel caso in cui la variabile *notice* assuma valore diverso da -1 e 5, viene visualizzato un messaggio di warning. La variabile *notice* è un intero definito nel file *symbol\_table.h* che serve a stabilire quale warning occorre stampare a video.

Consideriamo ora in che modo vengono gestite le funzioni all'interno della grammatica.

```

expr:
 r_variable { check($1,index_element, yylineno, true); }
 | T_CONSTANT { check($1, 0, yylineno, true); }
 | expr_without_variable
;

```

Il non terminale *expr* sviluppa tre produzioni:

- nella prima fa riferimento ad una variabile in fase di lettura. In questo caso occorre semplicemente verificare che tale variabile sia già stata precedentemente definita, e pertanto viene eseguita la funzione *check*;
- nella seconda fa riferimento a un costante, abbinata all'esecuzione della stessa funzione *check*;
- nella terza produzione viene impiegato il non terminale *expr\_without\_variable* che prevede un elevato numero di produzioni.

Di seguito vengono mostrate alcune delle più importanti regole di produzione del non terminale *expr\_without\_variable*.

**ASSEGNAZIONE**

```
T_CONSTANT '=' expr { isconstant($1, yylineno); yyerror("ERRORE
SEMANTICO: non è consentito assegnare un valore a una costante"); }
```

In questa produzione viene considerato il caso in cui si voglia assegnare una data espressione ad un valore costante rappresentato da una sequenza di caratteri. Naturalmente in tal caso si verifica un errore. Prima di visualizzare il messaggio di errore tramite *yyerror* viene eseguita la funzione *isconstant* presente nel file *symbol\_table.h* che si accerta del fatto che *T\_STRING* sia una costante, predefinita del PHP o definita dall'utente.

```
w_variable '=' expr
{
 if(array) {
 type_array_checking(T, 'c', NULL, yylineno);
 gen_create_array(f_ptr, $1, current_type, Exp);
 add_element($1, "array", current_type, NULL, dim,yylineno);
 array = false;
 } else {
 countelements(T) > 1 ? current_value = "0" :current_value;
 gen_assignment(f_ptr, 0, $1, current_type, index_element, Exp,
 false);
 add_element($1,"variable",current_type,current_value,0,yylineno);
 }
 clear();
}
```

Questa produzione rappresenta l'operazione di assegnazione di un'espressione a una variabile.

In questo caso la variabile è rappresentata da *w\_variable*, tale simbolo non terminale rappresenta quelle variabili a cui si assegna, mediante un'espressione semplice o complessa, un valore. Essa sarà anche scritta nella symbol table, se non è stata definita precedentemente. Il non terminale *w\_variable* fa riferimento alla seguente produzione:

```
w_variable:
 variable { $$ = $1; read = false; }
;
```

L'azione semantica associata assegna valore false al flag *read*.

Detto ciò, quando si effettua l'operazione di assegnazione, innanzitutto si verifica se *expr* è una definizione di array o no. Ciò viene fatto controllando il flag *array*.

Se si tratta di una definizione di array vengono eseguite le seguenti funzioni:

- *type\_array\_checking* con contesto 'c' ( create ): funzione presente nel file *symbol\_table.h* che ha il compito di controllare l'omogeneità dei tipi degli elementi dell'array mediante la lista concatenata *T*;
- *gen\_create\_array*: funzione presente nel file *gen\_code.h* che genera nel file di output la traduzione in codice C dell'istruzione di definizione di un array;
- *add\_element*: funzione presente nel file *symbol\_table.h* che ha il compito di scrivere l'array all'interno della symbol table.

Invece se si tratta di una variabile, vengono eseguite le seguenti operazioni:

- *countelements( T ) > 1 ? current\_value = "0" : current\_value*: se gli elementi di una lista concatenata *T* ( usata per il type checking ) sono in numero maggiore di uno, si attribuisce al valore della variabile valore 0, altrimenti si lascia invariato il valore di *current\_value*;
- *gen\_assignment*: funzione presente nel file *gen\_code.h* che genera nel file di output la traduzione in codice C dell'istruzione di assegnazione di una variabile;
- *add\_element*: funzione presente nel file *symbol\_table.h* che ha il compito di scrivere la variabile con il suo valore e il suo tipo all'interno della symbol table.

Infine viene eseguita la funzione *clear( )* di pulizia delle tre liste concatenate.

In maniera molto simile funziona la regola di produzione che gestisce l'assegnamento di una data espressione ad un elemento di un array.

```

element_array '='
{ fprintf(f_ptr, "%s[%s]", $1, index_element); check_index($1,
index_element, yylineno);}
expr {
check_element($1, index_element, yylineno, false);
gen_assignment(f_ptr, 0, $1, current_type, index_element, Exp, true);
clear();
}

```

Innanzitutto occorre stampare nel file di uscita il codice C dell'elemento dell'array. Dopodiché bisogna verificare che l'indice utilizzato per identificare quel dato elemento dell'array sia corretto. Ciò viene fatto tramite la funzione *check\_index* del file *symbol\_table.h*.

Successivamente vengono effettuati dei controlli tramite la funzione *check\_element* che ha il compito di controllare la correttezza dell'elemento dell'array e della espressione da assegnare all'elemento: è effettuato un controllo di tipi sulla base del contenuto della lista T.

Naturalmente il non terminale *element\_array* indica un elemento di un array e si sviluppa mediante la seguente regola sintattica.

```
element_array:
 T_VARIABLE '[' T_LNUMBER ']' { $$ = $1; index_element = $3; }
| T_VARIABLE '[' T_VARIABLE ']' { $$ = $1; index_element = $3; }
;
```

La struttura di un elemento di un array è rappresentata da un nome di variabile seguito da un intero o un'altra variabile (anch'essa di tipo intero) racchiusi tra parentesi quadre, che rappresentano l'indice dell'array.

Anche per le operazioni di assegnazione è possibile gestire la presenza di eventuali errori.

```
error '=' expr { yyerror("ERRORE SINTATTICO: parte sinistra
dell'espressione non riconosciuta"); }
```

In tal caso quando si verifica un errore sintattico sull'elemento a sinistra dell'operazione di assegnazione viene mostrato a video il messaggio di errore: *"ERRORE SINTATTICO: parte sinistra dell'espressione non riconosciuta"*.

### OPERAZIONE DI SOMMA

```
expr '+' { put_testo(&Exp, " + "); } expr { current_type =
type_checking(T, yylineno); }
```

All'interno di questa regola di produzione del non terminale *unticked\_statement* viene gestita l'operazione di somma tra due espressioni. La somma viene indicata dalla presenza del simbolo '+' posto tra un'espressione e l'altra. Nel momento in cui si individua la prima espressione seguita dal simbolo '+', viene immediatamente eseguita la funzione *put\_testo* che scrive per riferimento all'interno della lista concatenata di stringhe *Exp* il simbolo '+'. Questa lista serve a tener traccia degli elementi (variabili, elementi di un array o costanti) di un'espressione e delle operazioni che vengono impiegate. Successivamente si trova il non terminale *expr* che rappresenta l'elemento finale della produzione. Al termine dell'operazione di somma viene eseguita la funzione *type\_checking* del file *symbol\_table.h*. Questa funzione ha il compito di controllare i tipi degli



elementi che si vogliono sommare, fornendo dei warning nel caso in cui si utilizzino operandi di tipo differente tra loro e non conforme all'operazione di somma.

Anche per le operazioni di somma viene gestita la possibilità che si possano verificare degli errori.

```
expr '+'
{ yyerror("ERRORE SINTATTICO: (+) secondo termine dell'espressione mancante"); }
```

Con questa regola sintattica viene presa in considerazione la situazione di errore sintattico che si verifica quando un'operazione di somma termina con il simbolo '+' senza l'indicazione del successivo addendo. In tal caso, tramite la funzione *yyerror*, viene mostrato a video il messaggio *"ERRORE SINTATTICO: (+) secondo termine dell'espressione mancante"*.

Lo stesso tipo di meccanismo viene adottato per la trattazione delle altre operazioni aritmetiche quali  ***differenza (-)***

```
expr '-' { put_testo(&Exp, " - "); } expr { current_type =
type_checking(T, yylineno); }
| expr '-'
{ yyerror("ERRORE SINTATTICO: (-) secondo termine dell'espressione mancante"); }
```

### ***prodotto (\*)***

```
expr '*' { put_testo(&Exp, " * "); } expr { current_type =
type_checking(T, yylineno); }
| expr '*'
{ yyerror("ERRORE SINTATTICO: (*) secondo termine dell'espressione mancante "); }
```

### ***divisione (/)***

```
expr '/' { put_testo(&Exp, " / "); } expr { current_type =
type_checking(T, yylineno); }
| expr '/'
{ yyerror("ERRORE SINTATTICO: (/) secondo termine dell'espressione mancante"); }
```

### ***modulo (%)***

```
expr '%' { put_testo(&Exp, " % "); } expr { current_type =
type_checking(T, yylineno); }
| expr '%'
{ yyerror("ERRORE SINTATTICO: (%) secondo termine dell'espressione mancante"); }
```

**DEFINIZIONE DI UN ARRAY**

Un'altra produzione molto importante del non terminale *unticked\_statement* è rappresentata dalla definizione di un array.

```
T_ARRAY { array = true; dim = 0; } '(' array_pair_list ')'
```

Esistono diversi modi attraverso i quali è possibile definire un array in PHP. Nell'implementazione del progetto è stata prevista la modalità più utilizzata che presenta la seguente sintassi:

***\$nome\_array = array(elementi);***

Tale sintassi viene gestita dalla suddetta regola grammaticale.

Innanzitutto è necessaria la keyword *array* che viene passata dal lexer tramite il token *T\_ARRAY*. Una volta che il parser ha ricevuto questo token, capisce che si vuole procedere con la definizione di un array, e pertanto viene eseguita un'azione semantica che pone a *true* il flag *array* e setta a zero la variabile *dim* relativa alla dimensione dell'array. Dopodiché ci si aspetta di trovare il simbolo di parentesi tonda aperta e il non terminale *array\_pair\_list* seguito dal simbolo di parentesi tonda chiusa.

```
array_pair_list:
 /* empty */
 | non_empty_array_pair_list possible_comma
 ;
```

A sua volta, il non terminale *array\_pair\_list* può produrre un elemento vuoto, oppure una sequenza costituita dai non terminali *non\_empty\_array\_pair\_list* e *possible\_comma*.

```
non_empty_array_pair_list:
 non_empty_array_pair_list ',' { put_testo(&Exp, ",", " "); } scalar
 | scalar
 ;
```

Il non terminale *non\_empty\_array\_pair\_list* è rappresentato da uno scalare identificato dal non terminale *scalar*. Naturalmente, dato che un array può contenere più di un elemento, i vari elementi devono essere inseriti in successione separati da una virgola. Questa situazione viene prevista dalla prima derivazione di *non\_empty\_array\_pair\_list* che produce ricorsivamente se stesso seguito dal simbolo di virgola e dal non terminale *scalar*. Poiché occorre tener traccia degli elementi salvati all'interno dell'array, per poter poi effettuare tutti i relativi controlli di rito come, ad esempio, il controllo sull'omogeneità degli elementi ( grazie alla lista concatenata *T* ), viene eseguita la funzione *put\_testo* che inserisce all'interno della lista concatenata *Exp* il simbolo di virgola.

Come già detto in precedenza, il non terminale *scalar* fa riferimento ad un qualsiasi valore che può essere inserito all'interno di un array.

```
scalar:
 common_scalar
 | `` encaps_list ``
 | ``' encaps_list '\''
 ;
```

Infatti *scalar* prevede tali regole di produzione che sfociano nel non terminale *common\_scalar*, (che verrà esaminato successivamente), oppure nel già visto non terminale *encaps\_list* racchiuso tra doppi o singoli apici.

Infine, la regola di produzione del non terminale *array\_pair\_list* prevede anche la presenza del non terminale *possible\_comma*.

```
possible_comma:
 /* empty */
 | ','
 ;
```

Esso si traduce semplicemente nel simbolo della virgola, che può anche non essere presente.

### **COMMON\_SCALAR**

Andiamo ora a considerare il non terminale *common\_scalar* che fa riferimento all'acquisizione di un intero, un reale, una stringa, o una costante. Dal punto di vista prettamente grammaticale, questa regola è molto semplice:

```
common_scalar:
 T_LNUMBER
 | T_DNUMBER
 | T_CONSTANT_ENCAPSED_STRING
 | T_STRING
 ;
```

Si tratta semplicemente di ricevere uno di questi token da parte del lexer. A seconda del token ricevuto verrà acquisito un intero, un reale, una stringa racchiusa fra doppi apici o una stringa-costante senza apici.

Molto importante, invece, è l'azione semantica da associare ad ogni produzione.

```

common_scalar:
 T_LNUMBER {
 if(!read) {
 current_value = $1; current_type = "int";
 if(array || index_element != NULL) {
 dim++;
 }
 }
 put_testo(&T, "int");
 char *c = strdup($1, 1);
 if(strcmp(c, "-") == 0) {
 c = (char *)malloc((strlen($1) + 3) * sizeof(char));
 strcpy(c, "(");
 strcat(c, $1);
 strcat(c, ")");
 put_testo(&Exp, c);
 free(c);
 } else
 put_testo(&Exp, $1);
 }
| T_DNUMBER {
 if(!read) {
 current_value = $1; current_type = "float";
 if(array || index_element != NULL) {
 dim++;
 }
 }
 put_testo(&T, "float");
 char *c = strdup($1, 1);
 if(strcmp(c, "-") == 0) {
 c = (char *)malloc((strlen($1) + 3) * sizeof(char));
 strcpy(c, "(");
 strcat(c, $1);
 strcat(c, ")");
 put_testo(&Exp, c);
 free(c);
 } else
 put_testo(&Exp, $1);
}
| T_CONSTANT_ENCAPSED_STRING {
 if(!read) {
 current_value = $1; current_type = "char *";
 if(array || index_element != NULL) {
 dim++;
 }
 }
 put_testo(&T, "char *");
 put_testo(&Exp, $1);
}

```

```

| T_STRING {
 if(!read) {
 current_value = $1; current_type = "bool";
 if(array || index_element != NULL) {
 dim++;
 }
 }
 put_testo(&T, "bool");
 put_testo(&Exp, $1);
}
;

```

Considerando la prima produzione, quando il parser riceve il token *T\_LNUMBER* significa che è stato rilevato un numero intero.

Come azione semantica, viene subito effettuato un controllo sul valore del flag *read*. Esso specifica se il numero letto faccia parte di un espressione, anche complessa, di assegnazione a un variabile o a un elemento di un array ( appartiene a un'operazione di scrittura *read* = false ), oppure se fa parte di un espressione di controllo che, quindi, non è assegnata a nessun elemento ( appartiene a un'operazione di lettura *read* = true ). Nel caso in cui il flag è impostato a *false*, viene salvato il valore e il tipo dell'elemento all'interno delle variabili *current\_value* e *current\_type* (se è un'assegnazione semplice, il contenuto di *current\_value* sarà inserito nella Symbol table, altrimenti sarà inserito il valore zero). Dopodiché, se si tratta di un elemento di un array viene incrementata la dimensione dell'array stesso tramite la variabile *dim* (tale valore, dopo aver analizzato tutti gli elementi costituenti di un array, fornirà la sua dimensione). Dopo aver fatto questa serie di controlli, indipendentemente dai loro esiti, viene eseguita la funzione *put\_testo*. Questa funzione viene richiamata due volte:

- la prima volta ha il compito di inserire all'interno della lista concatenata *T* il tipo dell'elemento trattato. In questo caso verrà stampato il tipo intero, ossia *int*;
- la seconda volta ha il compito di inserire all'interno della lista concatenata *Exp* l'elemento stesso. I numeri interi o reali, se negativi, saranno prima racchiusi fra parentesi tonde e poi aggiunti alla lista *Exp*.

In questo modo, ogni volta che viene individuato un intero, si tiene traccia delle sue caratteristiche all'interno delle liste testo *T* e *Exp*. Queste liste verranno successivamente utilizzate per due scopi:

- la lista *T* sarà utilizzata per effettuare i controlli di tipo nelle funzioni *type\_checking* (per variabili) e *type\_checking\_array* definite nel file *symbol\_table.h*;

- la lista *Exp* sarà utilizzata per la traduzione. Essa contiene tutte le variabili, le costanti, gli elementi di un array, gli operatori facente parte di una espressione. Tale lista sarà utilizzata per la generazione e la stampa delle espressioni mediante le funzioni *gen\_expression*, *gen\_echo\_expression* e *print\_expression* definite nel file *gen\_code.h*.

Lo stesso tipo di meccanismo viene adottato all'interno delle parentesi graffe che indicano le azioni semantiche da eseguire per gli altri tre casi, ossia nel caso di un numero reale (*T\_DNUMBER*), una stringa (*T\_CONSTANT\_ENCAPSED\_STRING*) o una costante, o stringa senza apici, (*T\_STRING*).

## Sezione Epilogo

La sezione epilogo contiene tutte le routines C di supporto utili al corretto funzionamento del parser. In particolare sono state implementate le seguenti funzioni:

- *pulizia( )*;
- *yyerror( )*;
- *main( )*;

### *pulizia( )*

```
void pulizia() {
 chiudi_cancella_file(f_ptr);
}
```

La funzione *pulizia( )* viene utilizzata da parte della symbol table quando vengono eseguite delle funzioni di controllo che potrebbero portare alla chiusura del file di output.

La funzione *pulizia( )* richiama a sua volta la funzione *chiudi\_cancella\_file* che si trova nel file *gen\_code.h*. Questa funzione si occupa di chiudere ed eliminare il file di output di generazione del codice C, nel caso in cui si verifichi un errore semantico o un errore grave di altro tipo.

### *yyerror( )*

Come detto in precedenza *yyerror* è una funzione attraverso la quale il parser generator Bison è in grado di gestire gli errori che di verificano.

```
void yyerror(char* s) {
 _error++;
 printf("\033[01;31mRiga %d: %s.\n\033[00m", yylineno, s);
}
```

Il parametro che viene passato è una stringa che rappresenta l'errore da visualizzare. Quando si esegue la funzione viene aggiornato il numero di errori registrati incrementato il contatore degli errori `_error` e viene visualizzato il messaggio di errore indicando anche il numero di riga in corrispondenza del quale si è verificato.

### ***main()***

La funzione *main()* avvia il procedimento di lettura, parsing e traduzione dell'input. Rappresenta il cuore del traduttore, ossia la funzione principale a partire dalla quale vengono eseguite tutte le altre.

```
main(int argc, char *argv[]) {
 extern FILE *yyin;
 ++argv; --argc;
 for(int i = 0; i < argc; i++) {
 printf("Analisi del file %d: %s\n\n", i + 1, argv[i]);
 if(fopen(argv[i], "r") != NULL)
 yyin = fopen(argv[i], "r");
 else {
 printf("\033[01;31mERRORE: file %s non trovato.\033[00m\n",
 argv[i]);
 exit(0);
 }
 yyparse();
 printf("\n");
 }
 if(_error == 0 && _warning == 0) {
 printf("\n\n\033[01;32mParsing riuscito.\033[00m\n");
 } else if (_error == 0 && _warning != 0) {
 printf("\n\n\033[01;33mParsing riuscito, ma sono presenti %i
 warnings.\033[00m\n", _warning);
 } else {
 printf("\n\n\033[01;31mParsing fallito:\033[00m");
 if(_error > 1)
 printf("\033[01;31m sono stati rilevati %i errori.
 \033[00m\n", _error);
 else
 printf("\033[01;31m è stato rilevato %i errore. \033[00m\n",
 _error);
 cancella_file();
 }
}
```

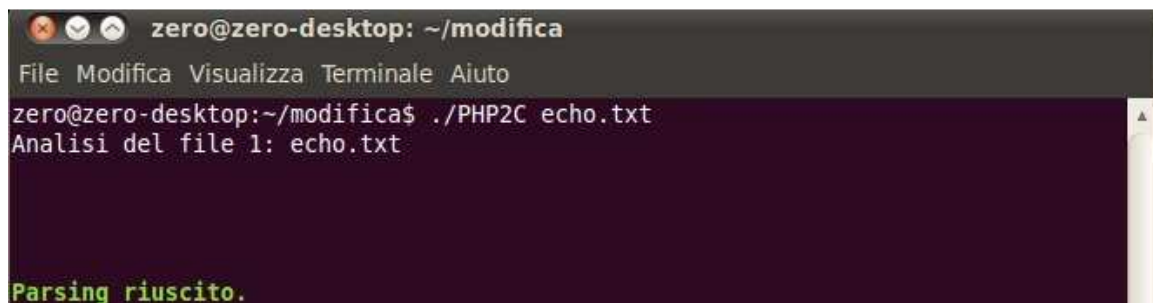
La funzione *main* passa due parametri: l'intero *argc* che indica il numero di file da esaminare e la stringa *argv* che contiene il nome del file o dei files che si vogliono analizzare.

Innanzitutto viene richiamato il puntatore al file sorgente *yyin* già definito all'interno del lexer.

Dopodiché, viene eseguito un ciclo per verificare se i file indicati esistono o meno nella cartella di appartenenza. Se un file non esiste viene visualizzato il messaggio di errore di *"file non trovato"* e interrotta tutta la procedura. Invece nel caso in cui il file sia effettivamente presente nella cartella, si procede con l'apertura del file e l'esecuzione della funzione di compilazione *yyparse()*.

Terminata la procedura di traduzione avviata mediante la funzione *yyparse()*, vengono effettuati dei controlli per stabilire quale messaggio visualizzare a video a conclusione dell'intero processo di traduzione. In particolare:

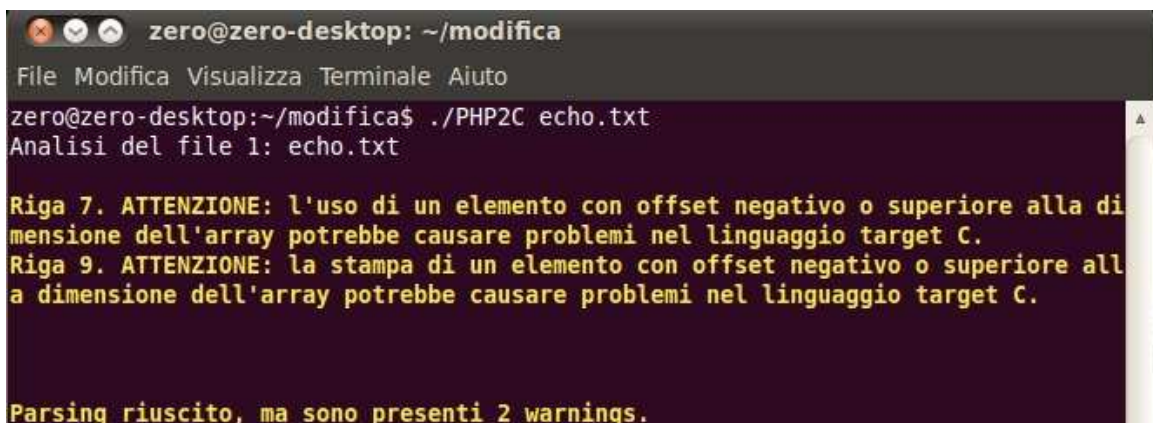
- se non sono stati rilevati né warning né errori, sarà visualizzata la stringa *"Parsing riuscito."*;



```
zero@zero-desktop: ~/modifica
File Modifica Visualizza Terminale Aiuto
zero@zero-desktop:~/modifica$./PHP2C echo.txt
Analisi del file 1: echo.txt

Parsing riuscito.
```

- se non sono stati rilevati errori ma soltanto warning, sarà visualizzata la stringa *"Parsing riuscito, ma sono presenti n warnings."* con l'indicazione del numero dei warnings;



```
zero@zero-desktop: ~/modifica
File Modifica Visualizza Terminale Aiuto
zero@zero-desktop:~/modifica$./PHP2C echo.txt
Analisi del file 1: echo.txt

Riga 7. ATTENZIONE: l'uso di un elemento con offset negativo o superiore alla dimensione dell'array potrebbe causare problemi nel linguaggio target C.
Riga 9. ATTENZIONE: la stampa di un elemento con offset negativo o superiore alla dimensione dell'array potrebbe causare problemi nel linguaggio target C.

Parsing riuscito, ma sono presenti 2 warnings.
```

- se è stato rilevato almeno un errore, sarà visualizzata la stringa *"Parsing fallito."* con l'indicazione del numero di errori trovati. In questo caso si procederà anche alla cancellazione del file di generazione del codice di uscita.





```
zero@zero-desktop: ~/modifica
File Modifica Visualizza Terminale Aiuto
zero@zero-desktop:~/modifica$./PHP2C echo.txt
Analisi del file 1: echo.txt

Riga 5. [FATALE] ERRORE SEMANTICO: l'uso di un operando di tipo non omogeneo i
n un array tipizzato non è corretto nel linguaggio target C.

Parsing fallito
```

## IMPLEMENTAZIONE DELLA SYMBOL TABLE

La *symbol table* è una struttura dati, utilizzata dal traduttore, che contiene un record per ogni identificatore presente nel codice sorgente del file d'ingresso. I vari campi del record rappresentano gli attributi dell'elemento. Questa struttura dati deve consentire di effettuare diverse operazioni su di essa, in particolare:

- verificare se un elemento con un determinato nome sia presente o meno nella symbol table;
- inserire un nuovo elemento all'interno della symbol table;
- aggiornare il valore di un elemento presente nella tabella;
- cancellare un elemento presente nella tabella.

Per la generazione della symbol table non esistono generatori automatici di codice. Pertanto tale struttura dati è stata realizzata implementando del codice C all'interno del file *symbol\_table.h*.

Innanzitutto sono state incluse alcune librerie molto utili per l'esecuzione delle funzioni che operano sugli elementi della symbol table.

```
#include <stdlib.h>
```

La libreria *stdlib.h* viene inclusa nel file in quanto contiene funzioni utili per le operazioni di allocazione della memoria nella tabella dei simboli.

```
#include <stdio.h>
```

La libreria *stdio.h* contiene le funzioni di input/output utili per la gestione dei messaggi di errore semantico da visualizzare a schermo.

```
#include <string.h>
```

La libreria *string.h* contiene funzioni utili per la gestione delle stringhe.

```
#include "utility.h"
```

Il file *utility.h* contiene una serie di funzioni e strutture di supporto al processo di compilazione e traduzione del codice sorgente in ingresso.

```
#include "gen_code.h"
```

Il file *gen\_code.h* contiene una serie di funzioni utili al processo di traduzione.

```
#include "uthash.h"
```

Il file *uthash.h* è una libreria di gestione degli indici HASH, utili per la gestione dei record della symbol table spiegata nel capitolo 05.

Vengono poi definite alcune costanti ed altri elementi utili per la gestione della symbol table.

```
#define NUM_CONSTANTS 3
```

*NUM\_CONSTANTS* indica la dimensione della tabella delle costanti predefinite del linguaggio PHP.

```
#define NUM_WARNINGS 6
```

*NUM\_WARNINGS* indica il numero di messaggi di warning che il compilatore può individuare durante il processo di compilazione.

```
void pulizia();
```

È la firma di una funzione definita nel file *php\_parser.h*: quando si verifica un errore semantico si procede con la chiusura e l'eliminazione del file di traduzione.

```
int notice = -1;
```

Si tratta di una variabile che, nel momento in cui viene sollevato un warning, serve ad identificare il tipo di warning che è stato rilevato impostando un determinato numero.

```
char* warn[NUM_WARNINGS] = { "ATTENZIONE: l'uso di un operando di
tipo stringa non è corretto nel linguaggio target C.\n",
"ATTENZIONE: l'uso di un operando di tipo boolean non è corretto nel
linguaggio target C.\n",
"ATTENZIONE: l'uso di un operando di tipo intero non è corretto nel
linguaggio target C.\n",
"ATTENZIONE: l'uso di un operando di tipo float non è corretto nel
linguaggio target C.\n",
"ATTENZIONE: l'uso di un elemento con offset negativo o superiore alla
dimensione dell'array potrebbe causare problemi nel linguaggio target
C.\n",
"ATTENZIONE: la stampa di un elemento con offset negativo o superiore
alla dimensione dell'array potrebbe causare problemi nel linguaggio
target C.\n" };
```

*warn* è un array di stringhe contenente tutti i possibili messaggi di warning che il compilatore può sollevare. L'accesso a uno specifico elemento dell'array è effettuato nel parser mediante l'indice *notice* inizialmente settato a -1.

Dopo queste dichiarazioni iniziali, si passa alla definizione delle strutture che rappresentano la vera e propria symbol table. In particolare vengono implementate due strutture:

- la tabella delle costanti (*CONSTANTS\_TABLE*);
- la Symbol Table vera e propria (*table*).

### **Tabella delle costanti**

```
typedef struct CONSTANTS_TABLE
{
 char *ctM;
 char *ctm;
} CONSTANTS_TABLE;
```

La tabella delle costanti è stata realizzata mediante la struttura *CONSTANTS\_TABLE* costituita da due stringhe:

- *ctM*: ossia il valore della costante espresso con lettere maiuscole;
- *ctm*: ossia il valore della costante espresso con lettere minuscole.

Sfruttando questa struttura, la tabella delle costanti è stata poi riempita con i valori di *true*, *false* e *null* tramite l'array *const\_tab*.

```
CONSTANTS_TABLE const_tab[NUM_CONSTANTS] = {
 { "TRUE", "true" },
 { "FALSE", "false" },
 { "NULL", "null" }
};
```

### **Symbol Table**

```
typedef struct {
 char *nameToken;
 char *element;
 char *type;
 char *value;
 int dim;
 UT_hash_handle hh;
} el_ST;
```

La struttura *el\_ST* definisce come deve essere strutturato ogni singolo record della symbol table.

In particolare ogni elemento della symbol table presenta i seguenti attributi:

- *nameToken*: stringa che indica il nome del token da inserire nella Symbol Table. Questo attributo funge da chiave per la struttura;
- *element*: stringa che serve a classificare il token come variabile (*variable*), come costante (*constant*) o come array (*array*);
- *type*: stringa utilizzata per indicare il tipo primitivo assunto dal token, che può essere *int*, *float*, *char \** o *bool*;
- *value*: stringa attraverso la quale viene indicato il valore della variabile. In caso di assegnazione semplice assume il valore assegnato alla variabile; in caso di assegnazione complessa assume valore pari a zero, nel caso in cui si tratti di un array assume valore *NULL*;
- *dim*: numero intero che indica la dimensione dell'array. È pari a zero nel caso di variabile o costante;
- *hh*: elemento che rende questa struttura indicizzabile mediante la libreria *uthash.h*.

Dopo aver definito la struttura della symbol table, sono stati definiti i puntatori attraverso i quali è possibile usufruire della tabella e dei suoi elementi.

```
typedef el_ST *element_ptr;
```

Puntatore al singolo elemento della symbol table.

```
element_ptr table = NULL;
```

Puntatore alla symbol table.

## GESTIONE DELLA SYMBOL TABLE

Nella restante parte del file sono state definite tutte le funzioni che consentono di interagire con gli elementi della symbol table e di operare su di essi.

### ***print\_elements ( )***

```
void print_elements() {
 element_ptr s;
 printf("\n/* * * * * * * * SYMBOL TABLE * * * * * * */\n\n");
 for(s = table; s != NULL; s = s->hh.next) {
 printf("element name %s element %s type %s value %s dim %i\n",
 s->nameToken, s->element, s->type, s->value, s->dim);
 }
 printf("\n/* * * * * * * * * * * * * * * * * */\n\n");
}
```

La funzione *print\_elements* stampa tutti gli elementi contenuti nella Symbol table, indicando per ciascuno di essi gli attributi *nameToken*, *element*, *type*, *value* e *dim*. Il ciclo di visualizzazione termina quando il puntatore all'elemento della symbol table assume valore *NULL*.

### ***find\_element ( )***

```
element_ptr find_element(char *nameToken) {
 element_ptr s;
 HASH_FIND_STR(table, nameToken, s);
 return s;
}
```

La funzione *find\_element* ha il compito di trovare e restituire un elemento presente nella Symbol table. Utilizza un solo parametro rappresentato dalla stringa *nameToken* che indica il nome del simbolo da ricercare. Per poter far ciò si avvale della funzione *HASH\_FIND\_STR* che si trova nel file *uthash.h* e che effettua la ricerca in maniera indicizzata tramite *HASH*.

La funzione *find\_element* restituisce l'elemento ricercato o il valore *NULL* nel caso in cui l'elemento ricercato non sia presente nella Symbol Table.

***delete\_element ( )***

```
void delete_element(element_ptr s) {
 HASH_DEL(table, s);
}
```

La funzione *delete\_element* ha il compito di eliminare un elemento dalla Symbol table.

L'argomento passato come parametro è il puntatore *s* all'elemento da rimuovere. Anche in questo caso, per far ciò, ci si avvale della funzione *HASH\_DEL* del file *uthash.h*.

***add\_element ( )***

```
void add_element(char *nameToken, char *element, char *current_type,
char *current_value, int dim, int nr) {
 element_ptr s;
 element_ptr exist = find_element(nameToken);
 if(!exist) {
 s = malloc(sizeof(el_ST));
 s->nameToken = nameToken;
 s->element = element;
 s->type = current_type;
 s->value = current_value;
 s->dim = dim;
 HASH_ADD_KEYPTR(hh, table, s->nameToken, strlen(s->nameToken), s);
 } else if (strcmp(exist->element, "constant") == 0) {
 printf("\033[01;31mRiga %i. [FATALE] ERRORE SEMANTICO:
ridefinizione di una costante.\033[00m\n", nr);
 printf("\n\n\033[01;31mParsing fallito.\033[00m\n");
 pulizia();
 exit(0);
 } else {
 if(strcmp(exist->type, current_type) == 0) {
 delete_element(exist);
 exist->value = current_value;
 HASH_ADD_KEYPTR(hh, table, exist->nameToken, strlen(exist->
nameToken), exist);
 } else {
 printf("\033[01;31mRiga %i. [FATALE] ERRORE SEMANTICO:
l'assegnazione viola il tipo primitivo della
variabile.\033[00m\n", nr);
 printf("\n\n\033[01;31mParsing fallito.\033[00m\n");
 pulizia();
 exit(0);
 }
 }
}
```

La funzione *add\_element* aggiunge un nuovo elemento nella Symbol table. Gli argomenti passati come parametri sono rappresentati dai seguenti sei elementi:

- *nameToken*: il nome del simbolo da aggiungere;
- *element*: la forma di elemento da aggiungere, ossia “constant”, “variable” o “array”;
- *current\_type*: il tipo primitivo dell’elemento, ossia “int”, “float” “char \*” o “bool”;
- *current\_value*: il valore assegnato alla variabile. Se si tratta di un’assegnazione complessa è pari a zero e se si tratta di un array è pari a NULL;
- *dim*: la dimensione dell’array. Se non si tratta di un array sarà pari a zero;
- *nr*: il numero riga segnalato dalla variabile *yylineno* di Flex.

Innanzitutto viene lanciata la funzione *find\_element* per verificare se il token sia presente o meno nella symbol table. Se non esiste, allora si procede con l’inserimento nella tabella tramite l’utilizzo della funzione *HASH\_ADD\_KEYPTR* del file *uthash.h*.

Se invece il token che si vuole inserire è presente nella symbol table ed è pari ad una costante, viene sollevato un errore fatale che blocca la compilazione in quanto si sta cercando di ridefinire una costante.

Se il token che si vuole inserire è presente nella symbol table ma non è una costante, significa che è stata effettuata un’operazione di riassegnazione del valore di una variabile o un elemento di un array. In tal caso occorre verificare che la riassegnazione non violi il tipo precedentemente definito per quella variabile o per quell’array. Nel caso in cui si verifichi la violazione del tipo viene lanciato a schermo l’errore semantico fatale “*ERRORE SEMANTICO: l’assegnazione viola il tipo primitivo della variabile*” che blocca la compilazione del codice. In caso contrario, solo nel caso di una variabile, si procede con l’aggiornamento degli attributi dell’elemento nella symbol table, cancellando l’elemento esistente ed inserendo il nuovo elemento.



***type\_checking ( )***

```
char *type_checking(testo *TT, int nr)
{
 testo *punt = TT;
 char *tipo;
 char *current_type = "int";
 while(punt != NULL) {
 if(strcmp(punt->tes, "float") == 0) {
 current_type = "float";
 }
 if(strcmp(punt->tes, "char *") == 0) {
 notice = 0;
 }
 if(strcmp(punt->tes, "bool") == 0) {
 notice = 1;
 }
 punt = punt->next;
 }
 return current_type;
}
```

La funzione *type\_checking* controlla se le varie operazioni matematiche previste dalla grammatica (+, -, \*, /, %, ++, --, <, <=, >, >=) abbiano come operandi numeri intere (*int*) o reali (*float*). Nel caso in cui tali operazioni abbiano come operandi delle stringhe o dei valori booleani, vengono generati determinati warning a seconda dei casi. In particolare ci saranno warnings differenti a seconda del valore assegnato alla variabile *notice*.

La funzione prevede due parametri:

- *TT*: una lista concatenata definita nel file *utility.h* ( lista T, "Tipi" ) che contiene tutti i tipi associati agli operandi;
- *nr*: il numero di riga segnalato dalla variabile *yylineno* di Flex.

La funzione *type\_checking* restituisce il tipo da assegnare alla variabile che si vuole aggiungere o aggiornare all'interno della Symbol table.

**type\_array\_checking ( )**

```

void type_array_checking(testo *TT, char context, element_ptr exist,
int nr) {
 testo *punt = TT;
 char *tipo;
 switch(context) {
 case 'c':
 if(TT != NULL)
 tipo = punt->tes;
 while(TT != NULL) {
 if(strcmp(TT->tes, tipo) != 0){
 printf("\033[01;31mRiga %i. [FATALE] ERRORE SEMANTICO:
l'uso di un operando di tipo non omogeneo in un array
tipizzato non è corretto nel linguaggio target C.
\033[00m\n", nr);
 printf("\n\n\033[01;31mParsing fallito\033[01;31m\033[00m
\n");
 pulizia();
 exit(0);
 }
 TT = TT->next;
 }
 break;
 case 's':
 if(strcmp(exist->type, TT->tes) != 0) {
 printf("\033[01;31mRiga %i. [FATALE] ERRORE SEMANTICO:
l'assegnazione viola l'omogeneità degli elementi
dell'array \"%s\".\033[00m\n", nr, exist->nameToken);
 printf("\n\n\033[01;31mParsing fallito\033[01;31m\033[00m\n");
 pulizia();
 exit(0);
 }
 break;
 case 'm':
 tipo = exist->type;
 while(TT != NULL) {
 if(strcmp(TT->tes, tipo) != 0){
 printf("\033[01;31mRiga %i. [FATALE] ERRORE SEMANTICO:
l'assegnazione viola l'omogeneità degli elementi
dell'array \"%s\".\033[00m\n", nr, exist->nameToken);
 printf("\n\n\033[01;31mParsing fallito\033[01;31m\033[00m\n");
 pulizia();
 exit(0);
 }
 TT = TT->next;
 }
 break;
 } }

```

La funzione *type\_array\_checking* effettua una serie di controlli sugli array per garantire una corretta compilazione del codice sorgente. In particolare si occupa di verificare che in fase di creazione di un array, gli elementi che lo costituiscono siano omogenei tra loro, ossia siano tutti dello stesso tipo. Inoltre, quando viene effettuata un'operazione di assegnazione a un elemento di un array, la funzione *type\_array\_checking* verifica che il tipo risultante dell'operazione di assegnazione sia omogeneo con il tipo dell'array, vale a dire sia dello stesso tipo.

La funzione *type\_array\_checking* utilizza quattro parametri:

- *TT*: una lista testo definita in *utility.h* che contiene tutti i tipi associati agli operandi;
- *context*: carattere che indica il contesto, ossia l'ambito di utilizzo all'interno del quale dovrà essere utilizzata la funzione. In particolare questo carattere può assumere uno dei seguenti valori:
  - '*c*': sta per "*create*" e fa riferimento alla creazione di un array. La funzione con tale contesto è richiamata da un'azione semantica di una regola del parser;
  - '*s*': sta per "*single*" e fa riferimento ad un'operazione di assegnazione di un solo valore ad un elemento di un array;
  - '*m*': sta per "*multiple*" e fa riferimento ad un'operazione di assegnazione multipla ad un elemento di un array. Le funzioni con contesto '*s*' e '*m*' sono richiamate dalla funzione *check\_element* spiegata più avanti;
- *exist*: indica l'elemento ricercato all'interno della symbol table. In tal caso equivale all'array di appartenenza dell'elemento;
- *nr*, il numero di riga segnalato dalla variabile *yylineno* di Flex.

Innanzitutto la funzione esamina il valore del carattere *context*. Dopodiché, a seconda del suo valore esegue uno dei controlli precedentemente descritti. In ogni caso, quando viene violata l'omogeneità del tipo relativo ad un elemento dell'array viene mostrato a video un determinato messaggio di errore semantico fatale, e viene interrotto il processo di compilazione.

***check\_element\_gen\_code ( )***

```
int check_element_gen_code(char *nameToken) {
 if(find_element(nameToken) != NULL)
 return 1;
 return 0;
}
```

La funzione *check\_element\_gen\_code* controlla se l'elemento che viene passato come parametro esiste nella Symbol table. La funzione restituisce l'intero 1 se l'elemento viene trovato, l'intero 0 in caso contrario.

Tale funzione è utilizzata nel processo di traduzione delle istruzioni di assegnazione (funzioni *gen\_create\_array* e *gen\_assignment* dichiarate nel file *gen\_code.h*):

- se si assegna un valore a una nuova variabile, e quindi non presente in tabella, si stamperà, nel file contenente la traduzione in C del codice sorgente, anche il tipo associato;
- altrimenti se si riassegna un valore ad una variabile già esistente in tabella, non si stamperà il tipo associato.

L'unico parametro passato dalla funzione è rappresentato dalla stringa *nameToken* che indica il nome del simbolo da cercare nella symbol table.

**check\_index ( )**

```

void check_index(char *nameToken, char *offset, int nr) {
 int index;
 element_ptr exist = find_element(nameToken);
 element_ptr exist_index;
 if(!exist) {
 printf("\033[01;31m\033[01;31mRiga %i. [FATALE] ERRORE
SEMANTICO: variabile \"%s\" non definita.\033[00m\n", nr, nameToken
);
 printf("\n\n\033[01;31mParsing fallito\033[01;31m\033[00m\n");
 pulizia();
 exit(0);
 } else {
 if(isnumeric(offset)) {
 index = atoi(offset);
 } else {
 exist_index = find_element(offset);
 if(exist_index) {
 if(strcmp(exist_index->type, "int") != 0) {
 printf("\033[01;31mRiga %i. [FATALE] ERRORE
SEMANTICO: l'uso di un elemento con offset non intero non è
ammissibile.\033[00m\n", nr);
 printf("\n\n\033[01;31mParsing
fallito\033[01;31m\033[00m\n");
 pulizia();
 exit(0);
 }
 index = atoi(exist_index->value);
 } else {
 printf("\033[01;31mRiga %i. [FATALE] ERRORE
SEMANTICO: variabile \"%s\" non definita.\033[00m\n", nr, offset);
 printf("\n\n\033[01;31mParsing
fallito\033[01;31m\033[00m\n");
 pulizia();
 exit(0);
 }
 }
 if(index < 0 || index >= exist->dim) {
 notice = 4;
 }
 }
}

```

La funzione *check\_index* viene richiamata in caso di assegnazione di un valore ad un elemento di un array (quindi in fase di scrittura) al fine di controllare se l'elemento e il suo indice (o *offset*), siano validi.

La funzione *check\_index* utilizza tre parametri come argomenti:

- *nameToken*: il nome dell'elemento dell'array;
- *offset*: l'indice dell'elemento;
- *nr*: il numero di riga segnalato dalla variabile *yylineno* di Flex.

Innanzitutto si verifica se l'elemento indicato è presente o meno all'interno della symbol table. In caso negativo viene visualizzato a video il messaggio di errore di variabile non definita ed interrotto il processo di compilazione.

Se, invece, l'elemento è presente all'interno della symbol table, vengono eseguiti una serie di controlli circa l'offset. In particolare si verifica se l'offset è rappresentato da un numero o meno. Questo controllo viene effettuato tramite la funzione *isnumeric* contenuta all'interno del file *utility.h*.

Se la stringa *offset* è rappresentata da un numero, il suo contenuto viene convertito nel corrispondente numero intero tramite la funzione *atoi*.

In caso contrario significa che l'offset è rappresentato da una variabile. Pertanto occorre verificare che tale variabile sia presente nella symbol table tramite la funzione *find\_element*. Se la variabile esiste e, poiché un array ammette esclusivamente un numero intero come indice, se il tipo dell'indice è diverso da un intero "*int*" viene visualizzato un messaggio di errore semantico fatale. In caso contrario viene utilizzata ancora una volta la funzione *atoi* per convertire il suo contenuto nel corrispondente valore intero, altrimenti viene mostrato a video un messaggio di errore semantico ed interrotta la compilazione.

Infine, l'ultimo controllo da effettuare sull'offset riguarda la sua dimensione. Infatti se l'indice è minore di 0 o maggiore della dimensione specificata nella Symbol table, viene visualizzato a video il messaggio di warning "*ATTENZIONE: l'uso di un elemento con offset negativo o superiore alla dimensione dell'array potrebbe causare problemi nel linguaggio target C*" tramite l'assegnazione di un determinato valore alla variabile *notice*.

**check\_element ( )**

```

element_ptr check_element(char *nameToken, char *offset, int nr, bool
read){
 int index;
 char *tipo;
 char *el_array;
 element_ptr exist = find_element(nameToken);
 element_ptr exist_index;
 if(!exist) {
 printf("\033[01;31m\033[01;31mRiga %i. [FATALE] ERRORE
SEMANTICO: variabile \"%s\" non definita.\033[00m\n", nr,
nameToken);
 printf("\n\n\033[01;31mParsing fallito\033[01;31m\033[00m\n");
 pulizia();
 exit(0);
 } else {
 if(((strcmp(exist->element, "variable") == 0) || (strcmp(
exist->element,"constant") == 0)) && read) {
 put_testo(&T, exist->type);
 put_testo(&Exp, exist->nameToken);
 } else {
 if(read) {
 put_testo(&T, exist->type);
 el_array = nameToken;
 strcat(el_array, "[");
 strcat(el_array, offset);
 strcat(el_array, "]");
 strcat(el_array, "\\0");
 put_testo(&Exp, el_array);
 }
 if(isnumeric(offset)) {
 index = atoi(offset);
 } else {
 exist_index = find_element(offset);
 if(exist_index) {
 if(strcmp(exist_index->type, "int") != 0) {
 printf("\033[01;31mRiga %i. [FATALE] ERRORE
SEMANTICO: l'uso di un elemento con offset non intero
non è ammissibile.\033[00m\n", nr);
 printf("\n\n\033[01;31mParsing fallito\033[00m\n");
 pulizia();
 exit(0);
 }
 index = atoi(exist_index->value);
 } else {

```

```

 printf("\033[01;31mRiga %i. [FATALE] ERRORE SEMANTICO:
variabile \"%s\" non definita.\033[00m\n", nr, offset);
 printf("\n\n\033[01;31mParsing fallito\033[00m\n");
 pulizia();
 exit(0);
 }
}
if(index < 0 || index >= exist->dim) {
 notice = 4;
}
if(countelements(T) != 0 && !read) {
 switch(countelements(T)) {
 case 1: type_array_checking(T, 's', exist, nr);
 break;
 default: type_array_checking(T, 'm', exist, nr);
 break;
 }
}
}
}
return exist;
}

```

La funzione *check\_element* ha il compito di verificare l'esistenza delle variabili, degli elementi di un array o delle costanti quando vengono utilizzate in determinate situazioni. In particolare questa funzione viene utilizzata per effettuare il controllo quando si ha a che fare con:

- composizione di un'espressione di assegnazione, anche abbastanza complessa;
- il costrutto *switch*, costrutti di diramazione *if*, cicli *for*, *while* e *do-while*;
- operatori di auto incremento *++* o auto decremento *--* (solo per le variabili o elementi di un array).

La funzione *check\_element* utilizza quattro parametri:

- *nameToken*: il nome della variabile o dell'elemento di un array;
- *offset*: l'indice dell'elemento dell'array;
- *nr*: il numero di riga segnalato dalla variabile *yylineno* di Flex;
- *read*: flag utilizzato per specificare se l'elemento viene analizzato in fase di lettura o scrittura (solo per elementi di un array).

Tale funzione restituisce l'elemento identificato dal *nameToken* trovato nella Symbol table.



Questa funzione è molto simile alla funzione precedente, ma viene richiamata in un elevato numero di punti differenti della grammatica.

In particolare:

- *nel caso di lettura di una variabile, un elemento di un array o una costante che sia un operando di un'espressione o di un costrutto.* In tal caso la funzione è richiamata con flag *read* impostato a *true*. Dopo aver effettuato il controllo all'interno della Symbol table e aver individuato l'elemento, il tipo associato viene inserito all'interno della lista *T* (*Tipi*);
- *nel caso di assegnazione di un'espressione ad un elemento di un array.* La funzione è richiamata dopo la creazione dell'espressione di assegnazione con flag *read* impostato a *false*. Poiché l'espressione è già stata generata, sarà possibile avere a disposizione la lista *T*. In base al numero di elementi presenti nella lista, verrà chiamata la funzione *type\_array\_checking* con *context* pari a *'s'* o *'m'* per effettuare un controllo sull'omogeneità dei tipi.

Per prima cosa la funzione *check\_element* verifica se l'elemento indicato è presente o meno all'interno della symbol table. In caso negativo viene visualizzato a video il messaggio di errore di variabile non definita ed interrotto il processo di compilazione.

Se, invece, l'elemento è presente all'interno della symbol table, si controlla il tipo di elemento che si sta esaminando. Se si tratta di una costante o di una variabile in sola lettura (*read = true*), il suo tipo viene aggiunto alla lista *T* e il suo nome alla lista *Exp*. Invece, nel caso in cui si tratta di un elemento di un array in sola lettura, nella lista *T* viene aggiunto il tipo dell'elemento e nella lista *Exp* viene inserito il nome dell'elemento, ossia il nome dell'array comprensivo di offset nella forma *nome\_array[offset]*.

Dopodiché si procede con l'analizzare l'*offset*, effettuando su di esso tutti i controlli che sono già stati previsti nella precedente funzione *check\_index*.

Infine, dopo questi controlli sull'*offset*, se la lista *T* non è vuota e si sta assegnando un valore all'elemento di un array, ossia si è in fase di scrittura (*read = false*), in base al numero di elementi presenti nella lista verrà chiamata la funzione *type\_array\_checking* con *context* pari a *'s'* o *'m'* per effettuare un controllo sull'omogeneità dei tipi.

**echo\_check ( )**

```

void echo_check(char *nameToken, char *offset, int nr){
 int index;
 char *tmp = ") ? \"true\" : \"false\"";
 char *el_array;
 element_ptr exist = find_element(nameToken);
 element_ptr exist_index;
 if(!exist) {
 printf("\033[01;31m\033[01;31mRiga %i. [FATALE] ERRORE
SEMANTICO: variabile \"%s\" non definita.\033[00m\n", nr,
nameToken);
 printf("\n\n\033[01;31mParsing fallito\033[01;31m\033[00m\n");
 pulizia();
 exit(0);
 } else {
 if(strcmp(exist->element, "array") == 0){
 if(offset == NULL) {
 printf("\033[01;31m\033[01;31mRiga %i. [FATALE] ERRORE
SEMANTICO: offset non definito.\033[00m\n", nr);
 printf("\n\n\033[01;31mParsing fallito\033[01;31m\033[00m\n");
 pulizia();
 exit(0);
 }
 if(isnumeric(offset)) {
 index = atoi(offset);
 } else {
 exist_index = find_element(offset);
 if(exist_index) {
 if(strcmp(exist_index->type, "int") != 0) {
 printf("\033[01;31mRiga %i. [FATALE] ERRORE
SEMANTICO: l'uso di un elemento con offset non intero
non è ammissibile.\033[00m\n", nr);
 printf("\n\n\033[01;31mParsing fallito\033[00m\n");
 pulizia();
 exit(0);
 }
 index = atoi(exist_index->value);
 } else {
 printf("\033[01;31m\033[01;31mRiga %i. [FATALE]
ERRORE SEMANTICO: variabile \"%s\" non
definita.\033[00m\n", nr, nameToken);
 printf("\n\n\033[01;31mParsing fallito\033[01;31m\033[00m\n");
 pulizia();
 exit(0);
 }
 }
 el_array = nameToken;
 strcat(el_array, "[");
 }
 }
}

```

```

 strcat(el_array, offset);
 strcat(el_array, "]);
 strcat(el_array, "\\0");
 if (strcmp(exist->type, "bool") == 0) {
 char *c = (char *)malloc((strlen(el_array) + strlen(
tmp) + 1) * sizeof(char));
 strcpy(c, "(");
 strcat(c, el_array);
 strcat(c, tmp);
 put_testo(&Exp, c);
 free(c);
 } else
 put_testo(&Exp, el_array);

 if(index < 0 || index >= exist->dim) {
 notice = 5;
 }
} else {
 if (strcmp(exist->type, "bool") == 0) {
 char *c = (char *)malloc((strlen(nameToken) + strlen(
tmp) + 1) * sizeof(char));
 strcpy(c, "(");
 strcat(c, nameToken);
 strcat(c, tmp);
 put_testo(&Exp, c);
 free(c);
 } else
 put_testo(&Exp, nameToken);
}
if(strcmp(exist->type, "int") == 0)
 put_testo(&Phrase, " %i ");
else if(strcmp(exist->type, "float") == 0)
 put_testo(&Phrase, " %f ");
else
 put_testo(&Phrase, " %s ");
}
}

```

La funzione *echo\_check* verifica se le variabili, le costanti o gli elementi di array specificati nell'istruzione *echo* esistono nella Symbol table.

La funzione *echo\_check* utilizza tre parametri come argomenti:

- *nameToken*: il nome della variabile, dell'elemento di un array o di una costante, contenuta nell'espressione associata all'istruzione *echo*;
- *offset*: l'indice dell'elemento di un array;
- *nr*: il numero di riga segnalato dalla variabile *yylineno* di Flex.

Anche in questo caso, per prima cosa la funzione verifica se l'elemento indicato è presente o meno all'interno della symbol table. In caso negativo viene visualizzato a video il messaggio di errore di variabile non definita ed interrotto il processo di compilazione. Se, invece, l'elemento è presente all'interno della symbol table e si tratta di un array, vengono effettuati tutti i soliti controlli sull'offset già visti nelle funzioni precedenti e in caso positivo si procede con l'inserimento dell'elemento dell'array all'interno della lista *Exp* nella forma di *nome\_array[offset]*. Se si tratta di una costante o una variabile si procede con il suo inserimento all'interno della lista *Exp*.

Infine, sulla base del tipo di variabile analizzata, viene assegnata alla lista testo *Phrase* (definita nel file *utility.h*) il giusto identificatore di variabili impiegato dal C nella funzione *printf*. Ciò viene fatto per agevolare la realizzazione della frase di traduzione in codice C che dovrà essere prodotta in fase di generazione del codice tramite la funzione *gen\_echo* contenuta all'interno del file *gen\_code.h*.

***isconstant ( )***

```

char *isconstant(char *string, int nr){
 int i;
 int trov = 0;
 char *current_type = "bool";
 for(i = 0; i < NUM_COSTANTS; i++){
 if(strcmp(string, cost_tab[i].ctM) == 0)
 trov = 1;
 if(strcmp(string, cost_tab[i].ctm) == 0)
 trov = 1;
 }
 if(trov == 0) {
 element_ptr exist = find_element(string);
 if(exist && (strcmp(exist->element, "constant") == 0)) {
 current_type = exist->type;
 trov = 1;
 }
 }
 if(trov == 0) {
 printf("\033[01;31mRiga %i. [FATALE] ERRORE SEMANTICO: stringa
 \"%s\" non riconosciuta.\033[00m\r\n", nr, string);
 printf("\n\n\033[01;31mParsing fallito\033[01;31m\033[00m\n");
 pulizia();
 exit(0);
 }
 return current_type;
}

```

La funzione *isconstant* verifica se la stringa fornita in ingresso (identificata dal token *T\_STRING*) è una costante o meno e restituisce il tipo della costante per effettuare i vari controlli di tipo (*type\_checking*). In particolare la funzione controlla se la stringa fornita è una costante predefinita del PHP o una costante definita dall'utente mediante la funzione *define*. In caso negativo, ossia se la stringa non è una costante, viene mostrato a video il messaggio di errore di stringa non riconosciuta e interrotta la compilazione.

La funzione *isconstant* utilizza come argomenti due soli parametri:

- *string*: la stringa da controllare;
- *nr*: il numero di riga segnalato dalla variabile *yylineno* di Flex.

## GENERAZIONE DEL CODICE C

Dopo aver trattato il processo di compilazione, relativo al sottoinsieme della porzione di linguaggio PHP preso in esame per la realizzazione del progetto, si è passati ad analizzare la fase di generazione del codice.

Questa fase rappresenta la parte vera e propria di traduzione, attraverso la quale il codice sorgente scritto in linguaggio PHP viene tradotto nel corrispondente codice target scritto in linguaggio C.

Come già visto in precedenza, per la generazione del codice sono state utilizzate una serie di funzioni che vengono richiamate dal parser al momento dell'esecuzione delle azioni semantiche previste nella grammatica.

Parte delle funzioni che fanno riferimento alla generazione del codice C sono state implementate all'interno del file *gen\_code.h*, la restante parte è presente, sottoforma di azione semantica, direttamente nella grammatica. Di seguito verrà descritta la struttura di questo file e le relative funzioni che lo costituiscono.

Nella parte dichiarativa del file, sono state effettuate una serie di operazioni preliminari come l'inclusione di librerie o la definizione di costanti e altri elementi, in maniera tale da dichiarare tutto ciò che potrà essere utile all'esecuzione delle varie funzioni.

```
#include <stdio.h>
```

La libreria *stdio.h* contiene le funzioni di input/output utili per la gestione dei messaggi di errore semantico da visualizzare a schermo.

```
#include <string.h>
```

La libreria *string.h* contiene funzioni utili per la gestione delle stringhe.

```
#define PATH "home/nomeutente/Scrivania/f_out.c"
```

*PATH* è una costante che rappresenta il percorso all'interno del quale dovrà essere salvato il file "*f\_out.c*" contenente la traduzione in C.

```
char *op_name[] = { "=", "+=", "-=", "*=", "/=", "%=" };
```

Si tratta di un array contenente tutti i possibili operatori di assegnazione.

## FUNZIONI DI GENERAZIONE DEL CODICE

Nella restante parte del file sono state definite tutte le funzioni che consentono di eseguire la generazione del codice C.

### ***apri\_file ( )***

```
FILE *apri_file() {
 FILE *f_ptr;
 if ((f_ptr = fopen(PATH, "w")) == NULL) {
 printf("\033[01;31mERRORE: apertura del file f_out.c fallita.
Directory %s non esistente.\033[00m\n", PATH);
 exit(0);
 }
 return f_ptr;
}
```

La funzione *apri\_file* serve ad aprire il file *f\_out.c* che conterrà il codice C generato in seguito al processo di compilazione e traduzione. Nel caso in cui il percorso indicato nella costante *PATH* non sia corretto, viene visualizzato a video il messaggio di errore di “*directory non esistente*” e viene interrotta sia la compilazione che la traduzione.

La funzione restituisce il puntatore al file *f\_out.c* di generazione del codice.

### ***chiudi\_file ( )***

```
void chiudi_file(FILE *f_ptr) {
 fflush(f_ptr);
 fclose(f_ptr);
}
```

La funzione *chiudi\_file* serve a chiudere il file *f\_out.c* che era stato precedentemente aperto. Utilizza come argomento il solo parametro *f\_ptr*, ossia il puntatore al file da chiudere.

**chiudi\_cancella\_file ( )**

```
void chiudi_cancella_file(FILE *f_ptr) {
 if(f_ptr != NULL)
 chiudi_file(f_ptr);
 if (remove(PATH) == -1)
 printf("\033[01;31mERRORE: impossibile cancellare il
 file.\033[00m\n");
}
```

La funzione *chiudi\_cancella\_file* ha il compito di chiudere ed eliminare il file *f\_out.c* nel caso in cui si verifichi un errore semantico o grave.

Utilizza come argomento il solo parametro *f\_ptr*, ossia il puntatore al file da chiudere.

Per eliminare il file ci si avvale della funzione *remove* che serve appunto a rimuovere files. Questa funzione ritorna un numero intero. In particolare:

- il numero zero indica che l'operazione di rimozione è andata a buon fine;
- il numero negativo -1 indica che si è verificato un errore.

Quando si verifica un errore nel processo di eliminazione del file viene visualizzato a video il messaggio *"ERRORE: impossibile cancellare il file"*.

**cancella\_file ( )**

```
void cancella_file() {
 FILE *f_ptr;
 if ((f_ptr = fopen(PATH, "r"))) {
 fclose(f_ptr);
 if (remove(PATH) == -1)
 printf("\033[01;31mERRORE: impossibile cancellare il
 file.\033[00m\n");
 }
}
```

La funzione *cancella\_file* elimina il file di generazione del codice *f\_out.c*. Questa funzione viene richiamata ad ogni inizio compilazione, in modo tale da poter cancellare il precedente file *f\_out.c*



eventualmente presente e poter ricreare un nuovo file *f\_out.c*. Se non è presente nessun file con questo nome all'interno della cartella indicata da *PATH*, non viene svolta alcuna operazione.

### ***gen\_header ( )***

```
void gen_header(FILE* f_ptr) {
 fprintf(f_ptr, "#include <stdio.h>\n");
 fprintf(f_ptr, "#include <string.h>\n\n");
 fprintf(f_ptr, "void main(void) {\n");
 fprintf(f_ptr, "\ttypedef enum { false, true } bool;\n");
}
```

La funzione *gen\_header* ha il compito di generare l'header del file *f\_out.c*.

Utilizza come argomento il solo parametro *f\_ptr*, ossia il puntatore al file da chiudere.

Tale intestazione consiste nelle seguenti operazioni:

- inclusione della libreria standard *stdio.h*;
- inclusione della libreria standard *string.h*;
- apertura del metodo principale *main* del programma C da realizzare;
- definizione di una struttura per identificare i dati di tipo boolean.

### ***gen\_constant ( )***

```
void gen_constant(FILE* f_ptr, char *nameConstant, char *type, char
*value) {
 if(strcmp(type, "char *") == 0)
 fprintf(f_ptr, "const char %s[] = %s;\n", nameConstant, value);
 else
 fprintf(f_ptr, "const %s %s = %s;\n", type, nameConstant, value);
}
```

La funzione *gen\_constant* si occupa di stampare nel file *f\_out.c* la dichiarazione delle costanti.

Utilizza quattro parametri:

- *f\_ptr*: il puntatore al file all'interno del quale stampare la traduzione;
- *nameConstant*: il nome della costante;

- *type*: il tipo della costante;
- *value*: il valore della costante.

È possibile notare che la funzione effettua prima un controllo per verificare se si tratta di una costante di tipo stringa o no. Questo perché, in caso affermativo, è necessario stampare la costante come array di caratteri, altrimenti si utilizza la notazione solita:

*const tipo nome\_costante = valore.*

### ***gen\_expression ( )***

```
char *gen_expression(testo *Exp) {
 char *expression = NULL;
 if(Exp != NULL) {
 expression = Exp->tes;
 Exp = Exp->next;
 }
 while(Exp != NULL) {
 strcat(expression, Exp->tes);
 Exp = Exp->next;
 }
 if(Exp != NULL) {
 strcat(expression, "\\0");
 }
 return expression;
}
```

La funzione *gen\_expression* si occupa di generare le espressioni che saranno stampate nel file *f\_out.c*, costituite dalla composizioni di variabili, elementi di array e costanti legate fra loro dagli operatori previsti dalla grammatica del parser. Restituisce la stringa che rappresenta l'espressione da stampare.

La funzione utilizza come argomento il solo parametro *Exp* che rappresenta la lista contenente i vari elementi che fanno parte delle espressioni.

La funzione non fa altro che esaminare la lista *testo Exp*, e concatenare tutti i suoi elementi all'interno della stringa *expression*, che verrà poi ritornata dalla funzione.

***gen\_echo\_expression ( )***

Simile alla funzione precedente, tranne la seguente istruzione aggiunta al ciclo while:

```
int elements = countelements(Exp);

while(Exp != NULL) {
 if(elements > 1)
 strcat(expression, ", ");
 strcat(expression, Exp->tes);
 Exp = Exp->next;
}
```

Tale funzione è richiamata nella funzione ***gen\_echo( )*** (l'ultima funzione spiegata in questo capitolo), e serve per generare l'insieme delle variabili da inserire nella corrispondente funzione di stampa del linguaggio target C. Nel caso in cui gli elementi contenuti nella lista testo siano maggiori di uno, essi saranno concatenati con il simbolo ",". Tale accortezza servirà nel gestire stampe multiple di variabili:

```
echo "Lista variabili: $var1, $var2";

printf("Lista variabili: %s, %s", var1, var1);
```

***print\_expression ( )***

```
void print_expression(FILE* f_ptr, testo *Exp) {
 char *expression;
 expression = Exp->tes;
 Exp = Exp->next;
 while(Exp != NULL) {
 strcat(expression, Exp->tes);
 Exp = Exp->next;
 }
 strcat(expression, "\0");
 fprintf(f_ptr, "%s", expression);
}
```

La funzione *print\_expression* si occupa di stampare le espressioni nel file *f\_out.c*. Tale funzione è principalmente utilizzata come azione semantica all'interno di specifiche regole della grammatica.

La funzione utilizza due parametri come argomenti:

- *f\_ptr*: il puntatore al file all'interno del quale stampare la traduzione;
- *Exp*: la lista contenente i vari elementi che fanno parte delle espressioni.

### ***gen\_create\_array ( )***

```
void gen_create_array(FILE* f_ptr, char *name_array, char *type,
testo *Exp) {
 char *expression = gen_expression(f_ptr, Exp);
 if (check_element_gen_code(name_array))
 fprintf(f_ptr, "%s[] = { %s }", name_array, expression);
 else
 fprintf(f_ptr, "%s %s[] = { %s }", type, name_array,expression);
}
```

La funzione *gen\_create\_array* si occupa di stampare nel file *f\_out.c* la dichiarazione di un array.

Utilizza quattro parametri:

- *f\_ptr*: il puntatore al file all'interno del quale stampare la traduzione;
- *name\_array*: il nome dell'array;
- *type*: il tipo dell'array;
- *Exp*: la lista testo contenente gli elementi delle espressioni.

All'interno di questa funzione viene richiamata la funzione *check\_element\_gen\_code* del file *symbol\_table.h* che controlla se l'elemento indicato è presente o meno nella symbol table. Infatti bisogna verificare se l'array è già stato definito in precedenza o meno.

Se l'array non è stato definito, viene stampato anche il tipo dell'array

*tipo nome\_array [ ] = {espressione};*

in caso contrario il tipo viene omesso

*nome\_array [ ] = {espressione};*

***gen\_assignment ( )***

```

void gen_assignment(FILE* f_ptr, int index, char *left_var, char
*type, char *index_element, testo *Exp, bool array) {
 char *expression = gen_expression(f_ptr, Exp);
 if(!array) {
 if (check_element_gen_code(left_var))
 fprintf(f_ptr, "%s %s %s", left_var, op_name[index],
 expression);
 else
 fprintf(f_ptr, "%s %s %s %s", type, left_var, op_name[index
], expression);
 } else {
 fprintf(f_ptr, " %s %s", op_name[index], expression);
 }
}

```

La funzione *gen\_assignment* si occupa di stampare nel file *f\_out.c* la dichiarazione di una variabile o di un elemento di un array.

Utilizza sette parametri:

- *f\_ptr*: il puntatore al file all'interno del quale stampare la traduzione;
- *index*: indice utilizzato per identificare il tipo di operatore di assegnazione impiegato. Questo numero intero viene utilizzato come indice dell'array *op\_name* definito nella parte dichiarativa dello stesso file *gen code.h*;
- *left\_var*: il nome della variabile o dell'elemento di un array che si trova a sinistra dell'operazione di assegnazione;
- *type*: il tipo della variabile;
- *index\_element*: l'offset corrispondente all'elemento di un array;
- *Exp*: la lista contenente gli elementi delle espressioni.
- *array*: flag che indica se si tratta di un array o meno.

**gen\_if ( )**

```
void gen_if(FILE* f_ptr, testo *Exp) {
 char *expression = gen_expression(f_ptr, Exp);
 fprintf(f_ptr, "if(%s) {\n", expression);
}
```

La funzione *gen\_if* genera l'istruzione condizionale *if*.

Utilizza due parametri come argomenti:

- *f\_ptr*: il puntatore al file all'interno del quale stampare la traduzione;
- *Exp*: la lista contenente i vari elementi che fanno parte delle espressioni.

**gen\_elseif ( )**

```
void gen_elseif(FILE* f_ptr, testo *Exp) {
 char *expression = gen_expression(f_ptr, Exp);
 fprintf(f_ptr, " else if(%s) {\n", expression);
}
```

La funzione *gen\_elseif* genera l'istruzione condizionale *else if*.

Utilizza due parametri:

- *f\_ptr*: il puntatore al file all'interno del quale stampare la traduzione;
- *Exp*: la lista contenente i vari elementi che fanno parte delle espressioni.

**gen\_while ( )**

```
void gen_while(FILE* f_ptr, testo *Exp) {
 char *expression = gen_expression(f_ptr, Exp);
 fprintf(f_ptr, "while(%s) {\n", expression);
}
```

La funzione *gen\_while* genera l'istruzione di ciclo *while*.

Utilizza due parametri:

- *f\_ptr*: il puntatore al file all'interno del quale stampare la traduzione;
- *Exp*: la lista contenente i vari elementi che fanno parte delle espressioni.

**gen\_switch ( )**

```
void gen_switch(FILE* f_ptr, testo *Exp) {
 char *expression = gen_expression(f_ptr, Exp);
 fprintf(f_ptr, "switch(%s) {\n", expression);
}
```

La funzione *gen\_switch* genera il costrutto *switch*.

Utilizza due parametri:

- *f\_ptr*: il puntatore al file all'interno del quale stampare la traduzione;
- *Exp*: la lista contenente i vari elementi che fanno parte delle espressioni.

**gen\_echo ( )**

```
void gen_echo(FILE* f_ptr, testo *Exp, testo *Phrase) {
 char *phrase = gen_expression(f_ptr, Phrase);
 char *expression = gen_echo_expression(f_ptr, Exp);
 if(phrase != NULL && expression != NULL)
 fprintf(f_ptr, "printf(\"%s\\n\", %s);\n", phrase, expression);
 else if(phrase != NULL)
 fprintf(f_ptr, "printf(\"%s\\n\");\n", phrase);
 else
 fprintf(f_ptr, "printf(%s);\n", expression);
}
```

La funzione *gen\_echo* si occupa di stampare nel file *f\_out.c* l'istruzione di stampa a video, che in PHP è svolta dalla funzione *echo* e in C è svolta dalla funzione *printf*.

La funzione *gen\_echo* utilizza tre parametri come argomenti:

- *f\_ptr*: il puntatore al file all'interno del quale stampare la traduzione;
- *Exp*: la lista contenente i vari elementi che fanno parte delle espressioni;
- *Phrase*: la lista contenenti gli elementi della frase da associare all'espressione.

**gen\_tab ( )**

```
void gen_tab(FILE* f_ptr, int ntab) {
 int i = 0;
 for(i = 0; i < ntab; i++)
 fprintf(f_ptr, "\t");
}
```

La funzione *gen\_tab* ha il compito di stampare il simbolo di tab “\t” nel file *f\_out.c* per ottenere una corretta indentazione del codice tradotto.

Ricordiamo che la chiamata a tale funzione e i vari incrementi o decrementi del contatore *ntab*, sono stati omessi nel capitolo nove per non appesantire troppo la trattazione. L’uso di tale funzione è possibile visionarla nel codice in appendice.



## ALCUNE FUNZIONI UTILI

Nella realizzazione del progetto finora descritto sono state utilizzate diverse funzioni che hanno consentito di poter eseguire la fase di compilazione e traduzione del codice sorgente PHP.

A seconda dell'ambito di appartenenza, tali funzioni sono state incluse all'interno dei diversi file utilizzati sinora per la realizzazione del lavoro.

Tuttavia, poiché alcune funzioni venivano utilizzate anche in ambiti differenti, è stato deciso di raggrupparle ed inserirle all'interno di un nuovo file chiamato *utility.h*.

Questo file viene richiamato all'interno del file di definizione della symbol table (*symbol\_table.h*).

Pertanto il file *utility.h* e le sue funzioni possono essere utilizzate direttamente o indirettamente dalla symbol table, dal parser e dal generatore di codice.

Di seguito verrà descritta la struttura del file *utility.h* e le relative funzioni che lo costituiscono.

Nella parte dichiarativa del file, sono state effettuate una serie di operazioni preliminari come la definizione di strutture e puntatori, utili per l'esecuzione delle varie funzioni.

```
typedef enum { false, true } bool;
```

Definisce la struttura di un nuovo tipo di dato: *bool*. Questo tipo viene implementato per poter rappresentare i valori booleani.

```
typedef struct testo {
 char *tes;
 struct testo *next;
} testo;
```

Definisce la struttura per un nuovo tipo di dato: *testo*<sup>3</sup>. Si tratta di una lista concatenata di stringhe.

```
testo *T;
```

Lista concatenata contenente i tipi di variabili, elementi di array o costanti. Questa lista viene utilizzata molto per la gestione del *type\_checking*.

---

<sup>3</sup> Suggerimento carpito dalla relazione *IMPLEMENTAZIONE DI UN COMPILATORE ADA 95 E TRADUZIONE ADA95 - C* di Molendini Vincenzo e Trigiante Giuseppe. A.A. 2010/2011.

```
testo *Exp;
```

Lista concatenata contenente gli elementi di un'espressione (variabili, elementi di array, costanti, operatori). Di grande utilità per effettuare la stampa di intere espressioni nel file d'uscita *f\_out.c* contenente la traduzione in C.

```
testo *Phrase;
```

Lista concatenata contenente una serie di frasi e/o parole (non si tratta delle keyword, ma di semplici caratteri testuali o di qualsiasi altro genere).

Questa lista viene utilizzata molto per effettuare la stampa di intere frasi, anche associate a delle espressioni, all'interno del file *f\_out.c*.

## FUNZIONI

Dopo questa prima parte, vengono trattate le varie funzioni di utilità che sono descritte di seguito.

### ***clear ( )***

```
void clear() {
 T = NULL;
 Exp = NULL;
 Phrase = NULL;
}
```

La funzione *clear* provvede semplicemente a cancellare il contenuto di ogni lista inizializzandole a *NULL*. Si tratta di una funzione molto importante in quanto, ogni volta che il parser valida una frase, è necessario resettare di volta in volta tutte le liste richiamando tale funzione.

**put\_testo ( )**

```

void put_testo(testo **TT,char *str){
 testo *punt,*t_el;
 t_el = (testo *)malloc(sizeof(testo));
 t_el->tes = (char *)strdup(str);
 t_el->next = NULL;
 if(*TT == NULL)
 *TT = t_el;
 else{
 punt = *TT;
 while(punt->next!=NULL)
 punt = punt->next;
 punt->next = t_el;
 }
}

```

La funzione *put\_testo* serve ad inserisce una data stringa all'interno di una determinata lista testo.

La funzione utilizza due soli parametri:

- *TT*: un doppio puntatore alla lista per consentire l'aggiunta di nuovi elementi;
- *str*: la stringa da inserire all'interno della lista. In particolare questa stringa sarà inserita all'interno del campo *tes* relativo al nuovo elemento che si vuole inserire nella lista.

**get\_testo ( )**

```

void get_testo(testo *TT, char *nome){
 testo *punt = TT;
 printf("/* * * * * * * * * * * %s * * * * * * * * * */\n\n", nome);
 while(punt != NULL) {
 printf("Elemento: %s\n", punt->tes);
 punt = punt->next;
 }
 printf("\n/* */\n\n");
}

```

La funzione *get\_testo* serve a stampare a video il contenuto di una lista, ossia il valore di tutti gli elementi contenuti in essa.

La funzione utilizza due soli parametri:

- *TT*: un puntatore alla lista;
- *nome*: una stringa d'utilità che serve ad etichettare la stampa, in modo tale da poter identificare più facilmente la lista.

Questa funzione, attualmente, non viene mai utilizzata all'interno del compilatore in quando né in fase di compilazione né in fase di traduzione è richiesta la visualizzazione della lista testo. Tuttavia, si tratta di una funzione molto importante da utilizzare in fase di testing e sviluppo del software, in quanto consente di comprendere meglio il comportamento di determinate funzioni ed effettuare una sorta di debugging delle stesse.

### ***countelements ( )***

```
int countelements(testo *T) {
 testo *punt = T;
 int value = 0;
 while(punt != NULL) {
 value++;
 punt = punt->next;
 }
 return value;
}
```

La funzione *countelements* ha il compito di contare il numero di elementi presenti all'interno della lista testo fornita in ingresso, e restituire un numero intero che rappresenta proprio il numero di elementi contati.

Utilizza un solo parametro rappresentato dal puntatore alla lista testo *T*.

***isnumeric ( )***

```
int isnumeric(char *str){
 char *c = (char *)strndup(str, 1);
 if(strcmp(c, "-") == 0) {
 c = (char *)strdup(str + 1);
 while(*c){
 if(!isdigit(*c))
 return 0;
 c++;
 }
 } else {
 while(*str){
 if(!isdigit(*str))
 return 0;
 str++;
 }
 }
 return 1;
}
```

La funzione *isnumeric* serve a stabilire se la stringa fornita in ingresso è pari ad un numero o no. In caso affermativo restituisce l'intero 1, mentre in caso contrario restituisce 0.

La funzione utilizza un solo parametro rappresentato dalla stringa *str* da analizzare.

Per far ciò la funzione analizza la stringa carattere per carattere. Inizialmente viene effettuato un controllo sul primo carattere per verificare il segno dell'eventuale numero. Se viene riscontrato il simbolo "-", vuol dire che si tratta di un numero negativo. Pertanto vengono analizzati tutti gli altri caratteri controllando se ognuno di essi è un numero o meno tramite la funzione *isdigit*. Se anche un solo carattere non è rappresentato da un numero, la funzione si interrompe ritornando l'intero 0. Invece, nel caso in cui il primo carattere non è rappresentato dal simbolo "-", si procede direttamente ad effettuare il controllo precedente su tutti i caratteri a partire dal primo verificando se si tratta o meno di un carattere numerico. In caso negativo, anche in questo caso, la funzione si interrompe ritornando l'intero 0.

Al termine di tutti i controlli, se la funzione non è mai stata interrotta significa che tutti i caratteri sono di tipo numerico tranne il primo che potrebbe essere un numero o il simbolo "-". Pertanto, trattandosi di un numero, positivo o negativo che sia, viene restituito l'intero 1.

## IL TRADUTTORE *PHP2C*

### STRUTTURA DEL TRADUTTORE *PHP2C*

Lo sviluppo del compilatore e i vari test sono stati effettuati in ambiente Unix-like mediante l'utilizzo dei tools *Flex* e *Bison* e del compilatore *GCC* (GNU Compiler Collection).

Il traduttore *PHP2C* si compone di sette file:

- *php\_lexer.l*: file che contiene le specifiche per lo scanner generator *Flex*;
- *php\_parser.y*: file che contiene le specifiche per il parser generator *Bison*;
- *symbol\_table.h*: file contenente tutte le procedure di gestione della symbol table e del type checking su variabili, elementi di array e costanti;
- *gen\_code.h*: file contenente parte delle procedure e istruzioni utili per la traduzione in C dei sorgenti in PHP;
- *utility.h*: file contenente un insieme di strutture dati e procedure di supporto alla compilazione e traduzione;
- *uthash.h*: libreria di supporto per la gestione degli indici HASH;
- *script* di generazione automatica del traduttore.

Le istruzioni utili per compilare i vari file ed ottenere il file eseguibile sono:

```
bison -d php_parser.y
gcc -c php_parser.tab.c
flex php_lexer.l
gcc -c lex.yy.c
gcc -o PHP2C php_parser.tab.o lex.yy.o -lm
```

Tali istruzioni sono state riportate all'interno dello script di generazione automatica.

Nello specifico, l'istruzione

```
bison -d php_parser.y
```

provvede alla generazione del parser. Viene utilizzata l'opzione *-d* per istruire Bison della generazione di un file header contenente i codici dei token associati alla grammatica. Questo file sarà poi incluso e utilizzato dallo scanner Flex. Bison genererà quindi il file *php\_parser.tab.c* contenente il codice C che è in grado di eseguire l'analisi sintattica. Tale file verrà compilato mediante il comando

```
gcc -c parser.tab.c
```

che, tramite l'opzione *-c*, istruisce il compilatore a creare il relativo file oggetto.

Con il comando

```
flex php_lexer.l
```

viene creato lo scanner e quindi il file *lex.yy.c*. Tale file è poi compilato tramite l'istruzione successiva

```
gcc -c lex.yy.c
```

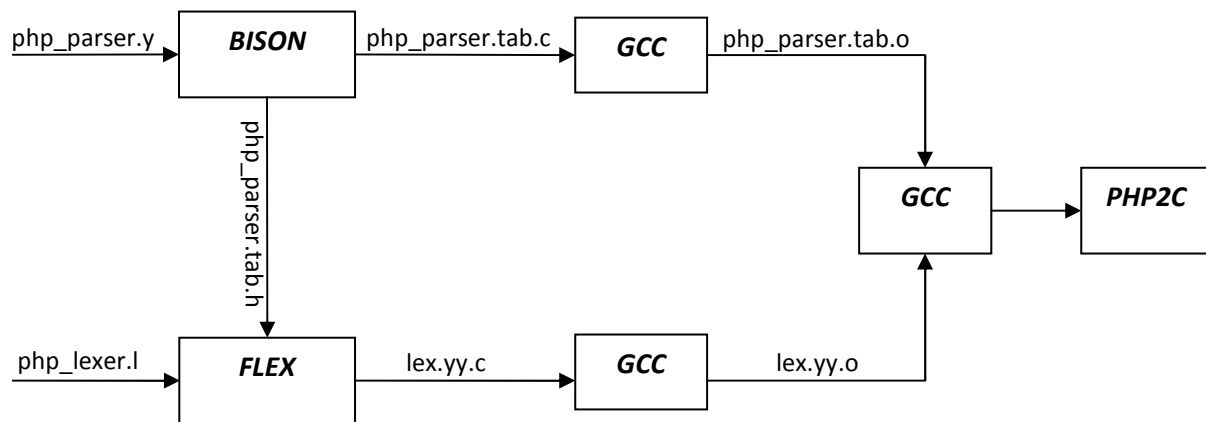
che, come nel caso precedente, si occupa di creare il relativo file oggetto.

Una volta ottenuti i diversi files oggetto, è necessario sottoporli ad un'operazione di *linkage* per la generazione del file eseguibile finale:

```
gcc -o PHP2C php_parser.tab.o lex.yy.o -lm
```

All'interno di questo comando, *PHP2C* rappresenta il nome del file eseguibile finale, mentre *php\_parser.tab.o* e *lex.yy.o* sono i files oggetto ottenuti in precedenza relativi, rispettivamente, al parser e al lexer. L'opzione *-lm* indica al compilatore di cercare le funzioni matematiche specificate nel codice.

Di seguito una sintesi grafica di come viene realizzato l'eseguibile *PHP2C*.



## ESECUZIONE DEL TRADUTTORE *PHP2C*

Come già detto in precedenza, per poter utilizzare il traduttore *PHP2C* è necessario operare in ambiente *Unix-like*.

Quindi, una volta che si hanno a disposizione tutti i file appena citati, è necessario inserirli tutti all'interno di una determinata directory a proprio piacimento.

Fatto ciò, la prima cosa da fare consiste nell'andare a modificare il contenuto del file *gen\_code.h*. In particolare bisogna modificare il contenuto della costante *PATH* che rappresenta il percorso all'interno del quale verrà salvato il file *f\_out.c* contenente la traduzione in C del file sorgente PHP. Per fare ciò è sufficiente aprire il file *gen\_code.h* con un qualsiasi editor di testo (come ad esempio *gedit*), e modificare la definizione della costante *PATH* al rigo 17. In particolare, data l'istruzione

```
#define PATH "/home/utente/traduttore/f_out.c"
```

è necessario modificare solo il messaggio posto tra doppi apici:

```
#define PATH "/home/utente/traduttore/f_out.c"
```



Se, ad esempio, si vuole salvare il contenuto della traduzione all'interno di un file di nome *programma.c* che si trovi sul desktop, ossia nella cartella */home/utente/Scrivania*, è necessario modificare il contenuto della costante *PATH* in questo modo:

```
#define PATH "/home/utente/Scrivania/programma.c"
```

Dopo aver svolto tale passaggio, è necessario salvare e chiudere il file *gen\_code.h*.

Ora si ha a disposizione tutto il necessario per poter eseguire il traduttore. Infatti è sufficiente lanciare lo script di generazione automatico del traduttore. Ciò può essere fatto manualmente, inserendo da riga di comando del *Terminale* le varie istruzioni contenute nello script, oppure automaticamente tramite la creazione di un lanciatore che esegua direttamente il contenuto dello script.

Una volta fatto ciò, all'interno della cartella nella quale si trovano tutti i file del traduttore, viene creato un nuovo file denominato *PHP2C* che rappresenta il vero e proprio file eseguibile.

Dopo aver ottenuto il file eseguibile, è possibile utilizzare il traduttore lanciandolo direttamente da linea di comando. Naturalmente occorre prima posizionarsi all'interno della directory che contiene il file *PHP2C*. Dopodiché è possibile digitare il seguente comando rappresentato dal nome del file eseguibile (*PHP2C*) e dal percorso del sorgente in codice PHP da tradurre.

```
./PHP2C nomefile.php
```

Naturalmente, se il file PHP si trova all'interno di un'altra cartella, è necessario indicare l'intero percorso di appartenenza. Ad esempio se il file si trova all'interno della directory */home/utente/scuola*, per poter effettuare la traduzione del file occorre lanciare il comando

```
./PHP2C /home/utente/scuola/nomefile.php
```

Una volta eseguito, il traduttore mostrerà a video gli eventuali errori di compilazione e warnings, e nel caso in cui non siano stati rilevati errori verrà generato il relativo file di traduzione in codice C.

Tutto il materiale relativo al programma *PHP2C* viene fornito sul *CD-ROM* in allegato.

Al suo interno, oltre ai file discussi sinora, sono stati inseriti anche alcuni file di testing che sono stati utilizzati sia in fase di sviluppo sia in fase di test per verificare il corretto funzionamento del software:

```
test_array.txt
test_array_def1.txt
test_array_def2.txt
test_array_op1.txt
test_array_op2.txt
test_constant.txt
test_echo.txt
test_if.txt
test_somma.txt
```

All'interno di un'altra cartella vengono infine inseriti una serie di semplici programmi PHP perfettamente funzionanti, sui quali è possibile testare il software *PHP2C*, ottenendo dei programmi C che sono in grado di svolgere la stessa funzione:

delta.php	=>	Calcola il determinante di un polinomio di secondo grado e stabilisce il tipo di radici;
fattoriale.php	=>	Calcola il fattoriale di un numero;
maggiore.php	=>	Individua l'elemento maggiore in un array;
media.php	=>	Calcola la media degli elementi di un array;
rettangolo.php	=>	Calcola l'area di un rettangolo;
temp.php	=>	Converte una temperatura da gradi Fahrenheit a Celsius.

## BIBLIOGRAFIA

1. Prof. Giacomo Piscitelli, *Dispense del corso di Linguaggi Formali e Compilatori*, a.a. 2010/2011.

2. Anthony A. Aaby , *Compiler Construction using Flex and Bison*, Walla Walla College

cs.wwc.edu - aabyan@wwc.edu, Version of February 25, 2004.

3. John Levine, *flex & bison*, O'Reilly, Version of August 2009.

4. The PHP Group. PHP: Hypertext Preprocessor. PHP: List of Parser Tokens – Manual.

[Online] <http://it.php.net/manual/en/tokens.php>.

5. The PHP Group. PHP: Hypertext Preprocessor. PHP: PHP Manual – Manual.

[Online] <http://it.php.net/manual/en/>.

6. Hanson, Troy. uthash: a hash table for C structures.

[Online] <http://uthash.sourceforge.net/index.html>.

# ***APPENDICE***

***PHP2C***

***php\_lexer.l***

```

1 %{
2
3 /*
4 +-----+
5 | PHP2C -- php_lexer.l |
6 +-----+
7 | Autori: BAVARO Gianvito |
8 | CAPURSO Domenico |
9 | DONVITO Marcello |
10 +-----+
11 */
12
13 #include <stdio.h>
14 #include "php_parser.tab.h"
15
16 %{
17
18 %x ST_IN_SCRIPTING
19 %x ST_DOUBLE_QUOTES
20 %x ST_SINGLE_QUOTE
21 %x ST_COMMENT
22 %x ST_DOC_COMMENT
23 %x ST_ONE_LINE_COMMENT
24 %option stack
25
26 %{
27 int lineno = 1;
28 //indica la riga dove è stato aperto un commento di tipo /* o /** che non è stato chiuso (si
29 veda la riga 499)
30 int comment_start_line = 0;
31 }
32
33 LNUM [0-9]+
34 NLNUM ("(-")[0-9]+(")")
35 DNUM ([0-9]*[\.][0-9]+)|([0-9]+[\.][0-9]*)
36 NDNUM ("(-")([0-9]*[\.][0-9]+(")")|("(-"[0-9]+[\.][0-9]*(")"))
37 CONST_LABEL "_"[A-Z][A-Z0-9_]*
38 LABEL [a-zA-Z_x7f-\xff][a-zA-Z0-9_x7f-\xff]*
39 WHITESPACE [\n\r\t]+
40 TABS_AND_SPACES [\t]*
41 TOKENS [;,:.\[\]\(\)|^&+-/*=%!~$<>?@]
42 ENCAPSED_TOKENS [\[\]\{\}\$]
43 ESCAPED_AND_WHITESPACE [\n\t\r #' .; , ()|^&+-/*=%!~<>?@]+
44 ANY_CHAR (.|[\n])
45 NEWLINE ("\r"|"n"|"r\n")
46
47 %option yylineno
48 %option noyywrap
49
50 %%
51
52
53 <ST_IN_SCRIPTING>"if" {
54 return T_IF;
55 }
56
57 <ST_IN_SCRIPTING>"elseif" {
58 return T_ELSEIF;
59 }
60
61 <ST_IN_SCRIPTING>"else" {
62 return T_ELSE;
63 }
64
65 <ST_IN_SCRIPTING>"while" {
66 return T_WHILE;
67 }
68
69 <ST_IN_SCRIPTING>"do" {
70 return T_DO;
71 }
72
73 <ST_IN_SCRIPTING>"for" {

```

```
74 return T_FOR;
75 }
76
77 <ST_IN_SCRIPTING>"switch" {
78 return T_SWITCH;
79 }
80
81 <ST_IN_SCRIPTING>"case" {
82 return T_CASE;
83 }
84
85 <ST_IN_SCRIPTING>"default" {
86 return T_DEFAULT;
87 }
88
89 <ST_IN_SCRIPTING>"break" {
90 return T_BREAK;
91 }
92
93 <ST_IN_SCRIPTING>"continue" {
94 return T_CONTINUE;
95 }
96
97 <ST_IN_SCRIPTING>"echo" {
98 return T_ECHO;
99 }
100
101 <ST_IN_SCRIPTING>"define" {
102 return T_DEFINE;
103 }
104
105 <ST_IN_SCRIPTING>"array" {
106 return T_ARRAY;
107 }
108
109 <ST_IN_SCRIPTING>"++" {
110 return T_INC;
111 }
112
113 <ST_IN_SCRIPTING>"--" {
114 return T_DEC;
115 }
116
117 <ST_IN_SCRIPTING>"==" {
118 return T_IS_EQUAL;
119 }
120
121 <ST_IN_SCRIPTING>"!="|"<" {
122 return T_IS_NOT_EQUAL;
123 }
124
125 <ST_IN_SCRIPTING>"<=" {
126 return T_IS_SMALLER_OR_EQUAL;
127 }
128
129 <ST_IN_SCRIPTING>">=" {
130 return T_IS_GREATER_OR_EQUAL;
131 }
132
133 <ST_IN_SCRIPTING>"+=" {
134 return T_PLUS_EQUAL;
135 }
136
137 <ST_IN_SCRIPTING>"-=" {
138 return T_MINUS_EQUAL;
139 }
140
141 <ST_IN_SCRIPTING>"*=" {
142 return T_MUL_EQUAL;
143 }
144
145 <ST_IN_SCRIPTING>"/=" {
146 return T_DIV_EQUAL;
147 }
```

```

148
149 <ST_IN_SCRIPTING>"%=" {
150 return T_MOD_EQUAL;
151 }
152
153 <ST_IN_SCRIPTING>"||" {
154 return T_BOOLEAN_OR;
155 }
156
157 <ST_IN_SCRIPTING>"&&" {
158 return T_BOOLEAN_AND;
159 }
160
161 <ST_IN_SCRIPTING>"OR" {
162 return T_LOGICAL_OR;
163 }
164
165 <ST_IN_SCRIPTING>"AND" {
166 return T_LOGICAL_AND;
167 }
168
169 <ST_IN_SCRIPTING>{TOKENS} {
170 return yytext[0];
171 }
172
173 <ST_IN_SCRIPTING>"{" {
174 yy_push_state(ST_IN_SCRIPTING);
175 return '{';
176 }
177
178 <ST_IN_SCRIPTING>"}" {
179 /* This is a temporary fix which is dependant on flex and it's implementation */
180 if (yy_start_stack_ptr) {
181 yy_pop_state();
182 }
183 return '}';
184 }
185
186 <ST_IN_SCRIPTING>{LNUM} {
187 yylval.id = (char *)strdup(yytext);
188 return T_LNUMBER;
189 }
190
191 <ST_IN_SCRIPTING>{NLNUM} {
192 //elimina le parentesi tonde, fra le quali è compreso il numero intero negativo.
193 char *s = (char *)malloc((strlen(yytext) - 1) * sizeof(char));
194 int j = 0;
195 int i;
196 for(i = 1; i < strlen(yytext) - 1; i++) {
197 s[j]=yytext[i];
198 j++;
199 }
200 s[strlen(s)] = '\0';
201 yylval.id = s;
202 return T_LNUMBER;
203 }
204
205 <ST_DOUBLE_QUOTES>{LNUM}|{NLNUM} { /* treat numbers (almost) as strings inside encapsulated
strings */
206 //se è un numero negativo elimino le parentesi tonde.
207 char *s = (char *)malloc((strlen(yytext) - 1) * sizeof(char));
208 s = (char *)strndup(yytext, 1);
209 if(strcmp(s, "(") == 0) {
210 int j = 0;
211 int i;
212 for(i = 1; i < strlen(yytext) - 1; i++) {
213 s[j] = yytext[i];
214 j++;
215 }
216 s[strlen(s)] = '\0';
217 yylval.id = s;
218 } else
219 yylval.id = (char *)strdup(yytext);
220

```

```

221 return T_NUM_STRING;
222 }
223
224 <ST_IN_SCRIPTING>{DNUM} {
225 yylval.id = (char *)strdup(yytext);
226 return T_DNUMBER;
227 }
228
229 <ST_IN_SCRIPTING>{NDNUM} {
230 //elimina le parentesi tonde, fra le quali è compreso il numero reale negativo.
231 char *s=(char *)malloc((strlen(yytext) - 1) * sizeof(char));
232 int j = 0;
233 int i;
234 for(i = 1; i < strlen(yytext) - 1; i++) {
235 s[j] = yytext[i];
236 j++;
237 }
238 s[strlen(s)] = '\0';
239
240 yylval.id = s;
241 return T_DNUMBER;
242 }
243
244 <INITIAL>"<?php"([\t]|{NEWLINE}) {
245 if (yytext[yyleng-1] == '\n') {
246 lineno++;
247 }
248 BEGIN(ST_IN_SCRIPTING);
249 return T_INIT;
250 }
251
252 <ST_IN_SCRIPTING>(">"){NEWLINE}? {
253 BEGIN(INITIAL);
254 return T_FINAL;
255 }
256
257 <ST_IN_SCRIPTING,ST_DOUBLE_QUOTES>"${LABEL} {
258 yylval.id = (char *)strdup(yytext + 1);
259 return T_VARIABLE;
260 }
261
262 <ST_IN_SCRIPTING,ST_DOUBLE_QUOTES>"${0-9}{LABEL} {
263 //Nel caso in cui si digiti erroneamente un numero dopo il $, parte una correzione
 automatica.
264 printf("\033[01;34mRiga: %i. CORREZIONE LESSICALE: e' stato corretto il nome della
 variabile \"%s\".\033[00m\n", yylineno, (char *)strdup(yytext + 1));
265 yylval.id = (char *)strdup(yytext + 2);
266 return T_VARIABLE;
267 }
268
269 <ST_IN_SCRIPTING,ST_DOUBLE_QUOTES>"${0-9}[0-9]+{LABEL} {
270 //Nel caso in cui si digiti erroneamente più di un numero dopo il $, parte una
 correzione automatica.
271 printf("\033[01;34mRiga: %i. CORREZIONE LESSICALE: e' stato corretto il nome della
 variabile \"%s\".\033[00m\n", yylineno, (char *)strdup(yytext + 1));
272 char *str = (char *)strdup(yytext);
273 int num = 0;
274
275 while(*str)
276 {
277 //conta le cifre.
278 if(isdigit(*str))
279 num++;
280 str++;
281 }
282 //Salta il simbolo "$" e le cifre
283 yylval.id = (char *)strdup(yytext + 1 + num);
284
285 return T_VARIABLE;
286 }
287
288 <ST_IN_SCRIPTING,ST_DOUBLE_QUOTES>{CONST_LABEL} {
289 yylval.id = (char *)strdup(yytext);
290 return T_CONSTANT;

```



```

291 }
292
293
294 <ST_IN_SCRIPTING>{LABEL} {
295 yylval.id = (char *)strdup(yytext);
296 return T_STRING;
297 }
298
299 <ST_DOUBLE_QUOTES>{LABEL} {
300 yylval.id = (char *)strdup(yytext);
301 return T_STRING;
302 }
303
304 <ST_IN_SCRIPTING>{WHITESPACE} {
305 int i;
306 for (i = 0; i < yyleng; i++) {
307 if (yytext[i] == '\n') {
308 lineno++;
309 }
310 }
311 /* elimina gli spazi, le tabulazioni e i newline nel codice contando le righe */
312 }
313
314 <ST_IN_SCRIPTING>"#|"//" {
315 BEGIN(ST_ONE_LINE_COMMENT);
316 yymore();
317 }
318
319 <ST_ONE_LINE_COMMENT>"?"|">" {
320 yymore();
321 }
322
323 <ST_ONE_LINE_COMMENT>[^\n\r?%>]+{ANY_CHAR} {
324 switch (yytext[yyleng - 1]) {
325 case '?': case '%': case '>':
326 yless(yyleng-1);
327 yymore();
328 break;
329 case '\n':
330 lineno++;
331 /* intentional fall through */
332 default:
333 BEGIN(ST_IN_SCRIPTING);
334 }
335 }
336
337 <ST_ONE_LINE_COMMENT>{NEWLINE} {
338 BEGIN(ST_IN_SCRIPTING);
339 lineno++;
340 }
341
342 <ST_ONE_LINE_COMMENT>"?>" {
343 yymore();
344 }
345
346 <ST_IN_SCRIPTING>"/*"{WHITESPACE} {
347 comment_start_line = lineno;
348 BEGIN(ST_DOC_COMMENT);
349 yymore();
350 }
351
352 <ST_IN_SCRIPTING>"/*" {
353 comment_start_line = lineno;
354 BEGIN(ST_COMMENT);
355 yymore();
356 }
357
358 <ST_COMMENT,ST_DOC_COMMENT>[^*]+ {
359 yymore();
360 }
361
362 <ST_DOC_COMMENT>"*/" {
363 BEGIN(ST_IN_SCRIPTING);
364 }

```

```

365
366 <ST_COMMENT>"*/" {
367 BEGIN(ST_IN_SCRIPTING);
368 }
369
370 <ST_COMMENT,ST_DOC_COMMENT>"*" {
371 yymore();
372 }
373
374 <ST_IN_SCRIPTING>([']({CONST_LABEL})[']) {
375 //elimina i singoli apici, restituendo il nome della costante.
376 char *s = (char *)malloc((strlen(yytext) - 1)*sizeof(char));
377 int j = 0;
378 int i;
379 for(i = 1; i < strlen(yytext) - 1; i++) {
380 s[j] = yytext[i];
381 j++;
382 }
383 s[strlen(s)] = '\0';
384 yylval.id = s;
385 return T_CONSTANT;
386 }
387
388 <ST_IN_SCRIPTING>(["]({CONST_LABEL})["]) {
389 //elimina i doppi apici, restituendo il nome della costante.
390 char *s = (char *)malloc((strlen(yytext) - 1)*sizeof(char));
391 int j = 0;
392 int i;
393 for(i = 1; i < strlen(yytext) - 1; i++) {
394 s[j] = yytext[i];
395 j++;
396 }
397 s[strlen(s)] = '\0';
398 yylval.id = s;
399 return T_CONSTANT;
400 }
401
402 <ST_IN_SCRIPTING>(["]([^$ _ "\\]| ("\\.\\.\\.)) *["]) {
403 yylval.id = (char *)strdup(yytext);
404 return T_CONSTANT_ENCAPSED_STRING;
405 }
406
407 <ST_IN_SCRIPTING>([']([^\` "\\]| ("\\.\\.\\.)) *[']) {
408 //sostituisce ai singoli apici i doppi apici, questo per evitare problemi con la
409 definizione (parola chiave define) delle costanti.
410 char *s=(char *)malloc((strlen(yytext) - 1)*sizeof(char));
411 int j = 1;
412 s[0] = '\'';
413 int i;
414 for(i = 1; i < strlen(yytext) - 1; i++) {
415 s[j] = yytext[i];
416 j++;
417 }
418 s[j] = '\'';
419 s[strlen(s)] = '\0';
420 yylval.id = s;
421 return T_CONSTANT_ENCAPSED_STRING;
422 }
423
424 <ST_IN_SCRIPTING>["] {
425 BEGIN(ST_DOUBLE_QUOTES);
426 return '\\"';
427 }
428
429 <ST_IN_SCRIPTING>['] {
430 BEGIN(ST_SINGLE_QUOTE);
431 return '\'';
432 }
433
434 <ST_DOUBLE_QUOTES>{ESCAPED_AND_WHITESPACE} {
435 //elimina gli spazi bianchi, questo per evitare che in traduzione compaiano spazi
436 bianchi indesiderati.
437 char *s= (char *)strdup(yytext);

```

```

437 if(strcmp(s, " ") != 0)
438 yylval.id = s;
439 else
440 yylval.id = "";
441
442 return T_ENCAPSED_AND_WHITESPACE;
443 }
444
445 <ST_SINGLE_QUOTE>([^\|\\]|\\[^\|\\])+ {
446 return T_ENCAPSED_AND_WHITESPACE;
447 }
448
449 <ST_DOUBLE_QUOTES>[`]+ {
450 return T_ENCAPSED_AND_WHITESPACE;
451 }
452
453 <ST_DOUBLE_QUOTES>"$"^[^a-zA-Z\x7f-\xff]{ } {
454 if (yyleng == 2) {
455 yless(1);
456 }
457 return T_CHARACTER;
458 }
459
460 <ST_DOUBLE_QUOTES>{ENCAPSED_TOKENS} {
461 return yytext[0];
462 }
463
464 <ST_DOUBLE_QUOTES>"\\{" {
465 return T_STRING;
466 }
467
468 <ST_SINGLE_QUOTE>"\\'" {
469 return T_CHARACTER;
470 }
471
472 <ST_SINGLE_QUOTE>"\\\\" {
473 return T_CHARACTER;
474 }
475
476 <ST_DOUBLE_QUOTES>"\\\\" {
477 return T_CHARACTER;
478 }
479
480 <ST_DOUBLE_QUOTES>"\\"[0-7]{1,3} {
481 return T_CHARACTER;
482 }
483
484 <ST_DOUBLE_QUOTES>"\\x"[0-9A-Fa-f]{1,2} {
485 return T_CHARACTER;
486 }
487
488 <ST_DOUBLE_QUOTES>["] {
489 BEGIN(ST_IN_SCRIPTING);
490 return '\\"';
491 }
492
493 <ST_SINGLE_QUOTE>['] {
494 BEGIN(ST_IN_SCRIPTING);
495 return '\'';
496 }
497
498 <ST_COMMENT,ST_DOC_COMMENT><<EOF>> {
499 printf("\033[01;33mATTENZIONE: commento iniziato alla riga %d e non terminato.\033[00m\n", comment_start_line);
500 return 0;
501 }
502
503 <ST_IN_SCRIPTING,INITIAL,ST_DOUBLE_QUOTES,ST_SINGLE_QUOTE>{ANY_CHAR} {
504 printf("\033[01;33mATTENZIONE: carattere inatteso in input '%c' (ASCII=%d) state=%d",
505 yytext[0], yytext[0], YYSTATE);
506 }

```

***PHP2C***

***php\_parser.y***

```

1 %{
2 /*
3 +-----+
4 | PHP2C -- php_parser.y |
5 +-----+
6 | Autori: BAVARO Gianvito |
7 | CAPURSO Domenico |
8 | DONVITO Marcello |
9 +-----+
10 */
11
12 /*
13 * LALR conflitti shift/reduce e come essi sono stati risolti:
14 *
15 * - 4 conflitti shift/reduce a causa dell'ambiguità pendente fra le regole (corretta e con
errori) del costrutto if. Risolti mediante shift.
16 * - 2 conflitti shift/reduce a causa dell'ambiguità pendente sui costrutti elseif/else.
Risolti mediante shift.
17 * - 6 conflitti shift/reduce a causa delle assegnazioni, semplici o in forma compatta, di
valori a elementi di un array. Risolti mediante shift.
18 * - 1 conflitto shift/reduce a causa dell'ambiguità pendente fra le due regole (corretta e
con errori) del costrutto for. Risolto mediante shift.
19 * - 39 conflitti shift/reduce a causa dell'ambiguità (introdotta con le azioni semantiche
fprintf) pendente fra tutte le espressioni avente i
simboli '(' e ')'. Risolti mediante shift.
20
21 *
22 */
23
24 #include <stdlib.h>
25 #include <stdio.h>
26 #include <string.h>
27 #include "symbol_table.h"
28
29 #define YYDEBUG 1
30
31 element_ptr element;
32 void yyerror(char *s); /* Il prototipo della funzione yyerror per la
visualizzazione degli errori sintattici. */
33
34 extern int yylineno; /* Il numero riga segnalato dalla variabile yylineno di
Flex. */
35
36 char *current_value; /* Il valore corrente di una variabile o costante. */
37 char *current_type; /* Il tipo corrente di una variabile o costante. */
38 char *index_element = NULL; /* L'offset di un elemento di un array. */
39 int _error = 0, _warning = 0, dim = 0; /* Contatore di errori, di warnings e di elementi di un
array (che definiscono la sua dimensione.) */
40
41 int ntab = 0; /* Contatore dei simboli di tabulazione. */
42 FILE *f_ptr; /* Puntatore al file che conterrà la traduzione in C. */
43
44 bool array, read; /* Flag usati per discriminare la dichiarazione di un
array, una variabile in lettura o scrittura. */
45
46
47 /* La funzione check avvia il controllo di una variabile, costante o elemento di un array.
Gli argomenti sono:
48 - nameToken, il nome del simbolo da aggiungere;
49 - offset, l'indice dell'elemento;
50 - nr, il numero riga segnalato dalla variabile yylineno di Flex;
51 - read, specifica se l'elemento è analizzato in lettura o scrittura
(solo per elementi di un array). */
52
53 void check(char *nameToken, char *offset, int nr, bool read)
54 {
55 element = check_element(nameToken, offset, yylineno, read);
56 current_value = element->value;
57 }
58
59
60 %}
61
62 %expect 52
63
64 //Associazione degli operatori e dei Token
65 %left ','
66 %left T_LOGICAL_OR
67 %left T_LOGICAL_AND
68 %right T_PRINT
69 %left '=' T_PLUS_EQUAL T_MINUS_EQUAL T_MUL_EQUAL T_DIV_EQUAL T_CONCAT_EQUAL T_MOD_EQUAL

```

```

70 %left '?' ':'
71 %left T_BOOLEAN_OR
72 %left T_BOOLEAN_AND
73 %nonassoc <id> T_IS_EQUAL T_IS_NOT_EQUAL
74 %nonassoc '<' T_IS_SMALLER_OR_EQUAL '>' T_IS_GREATER_OR_EQUAL
75 %left '+' '-' '*' '/' '%'
76 %left '*' '/' '%'
77 %right '!'
78 %right '~' T_INC T_DEC
79 %token T_IF
80 %left T_ELSEIF
81 %left T_ELSE
82 %left T_ENDIF
83 %left ')'
84 %token <id> T_LNUMBER T_DNUMBER T_STRING T_VARIABLE T_CONSTANT T_NUM_STRING
T_ENCAPSED_AND_WHITESPACE T_CONSTANT_ENCAPSED_STRING
85 %token T_CHARACTER
86 %token T_ECHO
87 %token T_DO
88 %token T_WHILE
89 %token T_FOR
90 %token T_SWITCH
91 %token T_CASE
92 %token T_DEFAULT
93 %token T_BREAK
94 %token T_CONTINUE
95 %token T_ARRAY
96 %token T_DEFINE
97 %token T_WHITESPACE
98 %token T_INIT
99 %token T_FINAL
100
101 /* L'opzione %union consente di specificare una varietà di tipi di dato che sono usati dalla
variabile yylval in Flex. L'unico tipo specificato è la
102 stringa di caratteri. */
103 %union{
104 char *id;
105 }
106
107 %type <id> variable r_variable w_variable element_array encaps_var common_scalar;
108
109 %start start;
110
111 %error-verbose
112 %locations
113
114
115 %% /* Regole */
116
117 /* Ogni volta che viene letto il simbolo di inizio script PHP "<?php" viene cancellato il file
f_out.c, frutto di precedenti compilazioni, e viene creato e aperto in modalità scrittura il
nuovo file. Tutte le le funzioni che iniziano con gen_*, dichiarate nel file gen_code.h, hanno
il compito di stampare nel file costrutti o espressioni. */
118 start:
119 T_INIT { cancella_file(); f_ptr = apri_file(); gen_header(f_ptr); }
top_statement_list
120 ;
121
122 /* La funzione fprintf sono utilizzate per stampare nel file f_out.c, contenente la traduzione
dei sorgenti in PHP in C, parte dei costrutti, parentesi ecc. La funzione gen_tab e il contatore
ntab servono per stampare nel file i simboli tab "\t" */
123 end:
124 T_FINAL { fprintf(f_ptr, "\n"); chiudi_file(f_ptr); }
125 ;
126
127 top_statement_list:
128 top_statement_list top_statement
129 | /* empty */
130 ;
131
132 top_statement:
133 { fprintf(f_ptr, "\t"); } statement
134 ;
135

```

```

136 inner_statement_list:
137 inner_statement_list { fprintf(f_ptr, "\t"); } inner_statement
138 | /* empty */
139 ;
140
141 inner_statement:
142 { gen_tab(f_ptr, ntab); } statement
143 ;
144
145 statement:
146 unticked_statement
147 ;
148
149 /* Tale regola definisce i costrutti primari e le istruzioni principali del PHP. Solo per le
costanti è utilizzata la funzione add_element (presente nel file symbol_table.h) che consente
di aggiungere nuovi elementi nella tabella dei simboli, quando esse sono dichiarate mediante il
token T_DEFINE. Per tutti gli altri costrutti si utilizzano le funzioni fprintf. Nel caso di
espressioni, expr, e dell'istruzione echo, T_ECHO, viene gestita la visualizzazione dei messaggi
di warning. Molto spesso è richiamata la funzione clear per svuotare le liste concatenate T, Exp
e Phrase (tutte contenute nel file utility.h). */
150 unticked_statement:
151 '{' inner_statement_list { fprintf(f_ptr, "\t"); } '}'
152 | T_DEFINE '(' T_CONSTANT ',' common_scalar ')' { clear(); add_element($3,
"constant", current_type, $5, 0, yylineno); gen_constant(f_ptr, $3, current_type, $5); } ';'
153 | T_IF '(' expr ')' { gen_if(f_ptr, Exp); clear(); ntab++; } statement
elseif_list else_single
154 | T_IF '(' error ')' statement elseif_list else_single
155 { yyerror("ERRORE SINTATTICO: espressione nel costrutto IF non
accettata"); }
156 | T_IF expr ')' statement elseif_list else_single
157 { yyerror("ERRORE SINTATTICO: '(' mancante nel costrutto IF"); }
158 | T_WHILE '(' expr ')' { gen_while(f_ptr, Exp); clear(); } while_statement
{ fprintf(f_ptr, "}\n"); }
159 | T_WHILE '(' error ')' while_statement
160 { yyerror("ERRORE SINTATTICO: espressione nel costrutto WHILE non
accettata"); }
161 | T_WHILE expr ')' while_statement
162 { yyerror("ERRORE SINTATTICO: '(' mancante nel costrutto WHILE"); }
163 | T_DO { fprintf(f_ptr, "do {\n"); ntab++; } statement T_WHILE { ntab--; gen_tab
(f_ptr, ntab); fprintf(f_ptr, "} while("); } '(' expr ')' { print_expression(f_ptr, Exp);
fprintf(f_ptr, ");\n"); clear(); } ';'
164 | T_FOR
165 '(' { fprintf(f_ptr, "for("); }
166 for_expr
167 ';' { fprintf(f_ptr, " "); }
168 for_expr { print_expression(f_ptr, Exp); clear(); }
169 ';' { fprintf(f_ptr, " "); }
170 for_expr { clear(); }
171 ')' { fprintf(f_ptr, ") {\n"); }
172 for_statement { gen_tab(f_ptr, ntab); fprintf(f_ptr, "}\n"); clear
(); }
173 | T_FOR '(' error ';' error ';' error ')' for_statement
174 { yyerror("ERRORE SINTATTICO: un argomento del costrutto FOR non è
corretto"); }
175 | T_SWITCH '(' expr ')' { gen_switch(f_ptr, Exp); clear(); } switch_case_list
{ gen_tab(f_ptr, ntab); ntab--; fprintf(f_ptr, "}\n"); }
176 | T_SWITCH expr ')' switch_case_list
177 { yyerror("ERRORE SINTATTICO: '(' mancante nel costrutto SWITCH"); }
178 | T_BREAK ';' { fprintf(f_ptr, "break;\n"); }
179 | T_CONTINUE ';' { fprintf(f_ptr, "continue;\n"); }
180 | T_ECHO echo_expr_list ';' {
181 gen_echo(f_ptr, Exp, Phrase);
182 clear();
183 //Stampa gli avvisi se notice è uguale a 5 (avviso riservato proprio
alla funzione echo).
184 if(notice == 5) {
185 printf("\033[01;33mRiga %i. %s\033[00m", yylineno, warn
[notice]);
186 _warning++;
187 notice = -1;
188 }
189 }
190 | T_ECHO error ';'
191 { yyerror("ERRORE SINTATTICO: argomento della funzione ECHO

```

```

errato"); }
| expr ';' {
192 if(countelements(Exp) == 1)
193 print_expression(f_ptr, Exp);
194 clear();
195 fprintf(f_ptr, "\n");
196 //Stampa gli avvisi se notice è diverso da -1 e da 5 (5 è un avviso
197 riservato alla funzione echo).
198 if(notice != -1 && notice != 5) {
199 printf("\033[01;33mRiga %i. %s\033[00m", yylineno, warn
[notice]);
200 _warning++;
201 notice = -1;
202 }
203 };
204 | ';' /* empty statement */
205 | end
206 ;
207
208 for_statement:
209 { ntab++; } statement { ntab--; }
210 ;
211
212 switch_case_list:
213 '{' { ntab++; } case_list '}'
214 ;
215
216 /* La funzione print_expression (presente nel file symbol_table.h) stampa un'espressione
direttamente nel file f_out.c. */
217 case_list:
218 /* empty */
219 | case_list T_CASE { ntab++; gen_tab(f_ptr, ntab); fprintf(f_ptr, "case "); }
expr { print_expression(f_ptr, Exp); clear(); } case_separator { fprintf(f_ptr, "\n"); }
inner_statement_list { ntab--; }
220 | case_list T_DEFAULT { ntab++; gen_tab(f_ptr, ntab); fprintf(f_ptr,
"default"); } case_separator { fprintf(f_ptr, "\n"); } inner_statement_list { ntab--; }
221 ;
222
223 case_separator:
224 ':' { fprintf(f_ptr, ":"); }
225 | ';' { fprintf(f_ptr, ";"); }
226 ;
227
228 while_statement:
229 { gen_tab(f_ptr, ntab); ntab++; } statement { ntab--; }
230 ;
231
232 elseif_list:
233 /* empty */ { fprintf(f_ptr, "\n"); gen_tab(f_ptr, ntab); ntab--; fprintf
(f_ptr, "}"); }
234 | elseif_list T_ELSEIF '(' expr ')' { gen_elseif(f_ptr, Exp); clear(); ntab++; }
statement { fprintf(f_ptr, "\n"); gen_tab(f_ptr, ntab); ntab--; fprintf(f_ptr, "}"); }
235 ;
236
237 else_single:
238 /* empty */ { fprintf(f_ptr, "\n"); }
239 | T_ELSE { fprintf(f_ptr, " else {\n"); ntab++; } statement { fprintf(f_ptr,
"\n"); gen_tab(f_ptr, ntab); ntab--; fprintf(f_ptr, "}\n"); }
240 ;
241
242 /* Tale sezione amministra l'espressione associata all'istruzione PHP di stampa echo.
243 Per le variabili e gli elementi di un array, in sola lettura, e le costanti è richiamata la
funzione echo_check (presente nel file symbol_table.h) al fine di controllare l'esistenza
delle stesse nella symbol table.
244 In caso di elementi di un array sarà anche controllato l'offset associato. */
245 echo_expr_list:
246 ''' encaps_list '''
247 | T_CONSTANT { echo_check($1, 0, yylineno); }
248 | T_CONSTANT_ENCAPSED_STRING { put_testo(&Exp, $1); }
249 | r_variable { echo_check($1, index_element, yylineno); }
250 ;
251
252 for_expr:
253 /* empty */

```



```

254 | expr
255 ;
256
257 /* Tale regola definisce la creazione delle espressioni associate ai vari costrutti del PHP.
258 La prima parte definisce la regola di assegnazione a una variabile o ad un elemento di un array,
mentre la seconda parte definisce le varie operazioni matematiche o logiche. L'ultima parte
amministra l'operatore condizionale ternario <condizione> ? <istruzione1> : <istruzione2> e
l'istanza di array (T_ARRAY). Nelle operazioni di assegnazione si assume la variabile
sinistra, in sola scrittura (read = false), mentre, per determinati operatori come ++ o --, la
variabile associata si assume come in sola lettura (read = true). */
259 expr_without_variable:
260 T_CONSTANT '=' expr { isconstant($1, yylineno); yyerror("ERRORE SEMANTICO:
non è consentito assegnare un valore a una costante"); }
261 | w_variable '=' expr {
262 if(array) {
263 //nel caso si stia definendo un array (un
particolare caso di assegnazione, possibile solo con l'operatore =) è effettuato un
type_checking sull'omogeneità degli elementi dell'array. Se il controllo ha esito positivo
l'istruzione è stampata nel file f_out.c e l'array viene aggiunto nella Symbol table.
264 type_array_checking(T, 'c', NULL, yylineno);
265 gen_create_array(f_ptr, $1, current_type, Exp);
266 add_element($1, "array", current_type, NULL,
dim, yylineno);
267 array = false;
268 } else {
269 //nel caso si stia definendo una variabile è
effettuato un controllo sulla lista concatenata T (Tipi). Se il numero di elementi è pari a
uno, allora è un'assegnazione semplice quindi è conservato il valore di current_value;
altrimenti il valore è impostato a zero. L'istruzione di assegnazione è stampata nel file
f_out.c e la variabile è aggiunta nella Symbol table.
270 countelements(T) > 1 ? current_value = "0" :
current_value;
271 gen_assignment(f_ptr, 0, $1, current_type,
index_element, Exp, false);
272 add_element($1, "variable", current_type,
current_value, 0, yylineno);
273 }
274 clear();
275 }
276 }
277 | element_array '=' {
278 //nel caso si voglia assegnare un valore a un elemento di un array, per
prima cosa viene controllato l'indice dell'elemento, mediante la funzione check_index
(contenuta nel file symbol_table.h).
279 fprintf(f_ptr, "%s[%s]", $1, index_element); check_index($1,
index_element, yylineno); } expr {
280 //successivamente mediante la funzione check_element (posta nel
medesimo file) si effettua un type_checking. Se i controlli hanno esito positivo, l'istruzione
di assegnazione sarà stampata nel file f_out.c .
281 check_element($1, index_element, yylineno, false);
282 gen_assignment(f_ptr, 0, $1, current_type, index_element, Exp, true);
283 clear();
284 }
285 | error '=' expr { yyerror("ERRORE SINTATTICO: parte sinistra dell'espressione non
riconosciuta"); }
286 | w_variable T_PLUS_EQUAL expr {
287 countelements(T) > 1 ? current_value = "0" : current_value;
288 gen_assignment(f_ptr, 1, $1, current_type, index_element, Exp, false);
289 add_element($1, "variable", current_type, current_value, dim,
yylineno);
290 clear();
291 }
292 | element_array T_PLUS_EQUAL { fprintf(f_ptr, "%s[%s]", $1, index_element);
check_index($1, index_element, yylineno); } expr {
293 check_element($1, index_element, yylineno, false);
294 gen_assignment(f_ptr, 1, $1, current_type, index_element, Exp, true);
295 clear();
296 }
297 | w_variable T_MINUS_EQUAL expr {
298 countelements(T) > 1 ? current_value = "0" : current_value;
299 gen_assignment(f_ptr, 2, $1, current_type, index_element, Exp, false);
300 add_element($1, "variable", current_type, current_value, dim,
yylineno);
301 clear();

```

```

302 }
303 | element_array T_MINUS_EQUAL { fprintf(f_ptr, "%s[%s]", $1, index_element);
check_index($1, index_element, yylineno); } expr {
304 check_element($1, index_element, yylineno, false);
305 gen_assignment(f_ptr, 2, $1, current_type, index_element, Exp, true);
306 clear();
307 }
308 | w_variable T_MUL_EQUAL expr {
309 countelements(T) > 1 ? current_value = "0" : current_value;
310 gen_assignment(f_ptr, 3, $1, current_type, index_element, Exp, false);
311 add_element($1, "variable", current_type, current_value, dim,
yylineno);
312 clear();
313 }
314 | element_array T_MUL_EQUAL { fprintf(f_ptr, "%s[%s]", $1, index_element);
check_index($1, index_element, yylineno); } expr {
315 check_element($1, index_element, yylineno, false);
316 gen_assignment(f_ptr, 3, $1, current_type, index_element, Exp, true);
317 clear();
318 }
319 | w_variable T_DIV_EQUAL expr {
320 countelements(T) > 1 ? current_value = "0" : current_value;
321 gen_assignment(f_ptr, 4, $1, current_type, index_element, Exp, false);
322 add_element($1, "variable", current_type, current_value, dim,
yylineno);
323 clear();
324 }
325 | element_array T_DIV_EQUAL { fprintf(f_ptr, "%s[%s]", $1, index_element);
check_index($1, index_element, yylineno); } expr {
326 check_element($1, index_element, yylineno, false);
327 gen_assignment(f_ptr, 4, $1, current_type, index_element, Exp, true);
328 clear();
329 }
330 | w_variable T_MOD_EQUAL expr {
331 countelements(T) > 1 ? current_value = "0" : current_value;
332 gen_assignment(f_ptr, 5, $1, current_type, index_element, Exp, false);
333 add_element($1, "variable", current_type, current_value, dim,
yylineno);
334 clear();
335 }
336 | element_array T_MOD_EQUAL { fprintf(f_ptr, "%s[%s]", $1, index_element);
check_index($1, index_element, yylineno); } expr {
337 check_element($1, index_element, yylineno, false);
338 gen_assignment(f_ptr, 5, $1, current_type, index_element, Exp, true);
339 clear();
340 }
341 //di seguito, solo per gli operatori matematici verrà eseguito un controllo dei tipi
con, eventualmente, visualizzazione di warnings.
342 | variable { check($1, index_element, yylineno, true); } T_INC { put_testo
(&Exp, "++"); type_checking(T, yylineno); print_expression(f_ptr, Exp); }
343 | T_INC { put_testo(&Exp, "++"); } variable { check($3, index_element,
yylineno, true); type_checking(T, yylineno); print_expression(f_ptr, Exp); }
344 | variable { check($1, index_element, yylineno, true); } T_DEC { put_testo
(&Exp, "--"); type_checking(T, yylineno); print_expression(f_ptr, Exp); }
345 | T_DEC { put_testo(&Exp, "--"); } variable { check($3, index_element,
yylineno, true); type_checking(T, yylineno); print_expression(f_ptr, Exp); }
346 | expr T_BOOLEAN_OR { put_testo(&Exp, " || "); } expr
347 | expr T_BOOLEAN_OR
348 { yyerror("ERRORE SINTATTICO: (||) secondo termine dell'espressione
mancante"); }
349 | expr T_BOOLEAN_AND { put_testo(&Exp, " && "); } expr
350 | expr T_BOOLEAN_AND
351 { yyerror("ERRORE SINTATTICO: (&) secondo termine dell'espressione
mancante"); }
352 | expr T_LOGICAL_OR { put_testo(&Exp, " OR "); } expr
353 | expr T_LOGICAL_OR
354 { yyerror("ERRORE SINTATTICO: (OR) secondo termine dell'espressione
mancante"); }
355 | expr T_LOGICAL_AND { put_testo(&Exp, " AND "); } expr
356 | expr T_LOGICAL_AND
357 { yyerror("ERRORE SINTATTICO: (AND) secondo termine dell'espressione
mancante"); }
358 | expr T_IS_EQUAL { put_testo(&Exp, " == "); } expr
359 | expr T_IS_EQUAL

```

```

360 { yyerror("ERRORE SINTATTICO: (==) secondo termine dell'espressione
mancante"); }
361 | expr T_IS_NOT_EQUAL { put_testo(&Exp, " != "); } expr
362 | expr T_IS_NOT_EQUAL
363 { yyerror("ERRORE SINTATTICO: (!=) secondo termine dell'espressione
mancante"); }
364 | expr '<' { put_testo(&Exp, " < "); } expr { current_type = type_checking(T,
yylineno); }
365 | expr '<'
366 { yyerror("ERRORE SINTATTICO: (<) secondo termine dell'espressione
mancante"); }
367 | expr T_IS_SMALLER_OR_EQUAL { put_testo(&Exp, " <= "); } expr { current_type =
type_checking(T, yylineno); }
368 | expr T_IS_SMALLER_OR_EQUAL
369 { yyerror("ERRORE SINTATTICO: (<=) secondo termine dell'espressione
mancante"); }
370 | expr '>' { put_testo(&Exp, " > "); } expr { current_type = type_checking(T,
yylineno); }
371 | expr '>'
372 { yyerror("ERRORE SINTATTICO: (>) secondo termine dell'espressione
mancante"); }
373 | expr T_IS_GREATER_OR_EQUAL { put_testo(&Exp, " >= "); } expr { current_type =
type_checking(T, yylineno); }
374 | expr T_IS_GREATER_OR_EQUAL
375 { yyerror("ERRORE SINTATTICO: (>=) secondo termine dell'espressione
mancante"); }
376 | expr '+' { put_testo(&Exp, " + "); } expr { current_type = type_checking(T,
yylineno); }
377 | expr '+'
378 { yyerror("ERRORE SINTATTICO: (+) secondo termine dell'espressione
mancante"); }
379 | expr '-' { put_testo(&Exp, " - "); } expr { current_type = type_checking(T,
yylineno); }
380 | expr '-'
381 { yyerror("ERRORE SINTATTICO: (-) secondo termine dell'espressione
mancante"); }
382 | expr '*' { put_testo(&Exp, " * "); } expr { current_type = type_checking(T,
yylineno); }
383 | expr '*'
384 { yyerror("ERRORE SINTATTICO: (*) secondo termine dell'espressione
mancante"); }
385 | expr '/' { put_testo(&Exp, " / "); } expr { current_type = type_checking(T,
yylineno); }
386 | expr '/'
387 { yyerror("ERRORE SINTATTICO: (/) secondo termine dell'espressione
mancante"); }
388 | expr '%' { put_testo(&Exp, " % "); } expr { current_type = type_checking(T,
yylineno); }
389 | expr '%'
390 { yyerror("ERRORE SINTATTICO: (%) secondo termine dell'espressione
mancante"); }
391 | '(' { put_testo(&Exp, "("); } expr ')' { put_testo(&Exp, ")"); }
392 | expr '?'
393 | expr ':'
394 | expr
395 scalar
396 //tale flag discrimina se l'assegnazione a una variabile sia in realtà la definizione di
un array.
397 | T_ARRAY { array = true; dim = 0; } '(' array_pair_list ')'
398 ;
399
400 /* Tale regola definisce i vari numeri, interi o reali, stringhe (racchiuse fra singoli apici
'' o doppi apici "") e le stringhe (senza apici), potenziali costanti predefinite del PHP o
definite dall'utente. Si noti che se viene letto un valore in scrittura (assegnazione), esso e
il suo tipo sono memorizzati nelle variabili current_value e current_type. Inoltre se è in
acquisizione un array viene incrementata la sua dimensione. Sia in caso di scrittura che di
lettura il valore e il tipo sono rispettivamente aggiunti nelle liste T (Tipi, per il controllo
dei tipi mediante le funzioni type_checking e type_array_checking del file symbol_table.h) e
Exp (Espressione, per la generazione di espressioni mediante le funzioni gen_expression,
gen_echo_expression o print_expression del file gen_code.h). */
401 common_scalar:
402 T_LNUMBER {
403 if(!read) {
404 current_value = $1; current_type = "int";

```

```

405 if(array || index_element != NULL) {
406 dim++;
407 }
408 }
409 put_testo(&T, "int");
410 //Se è un numero negativo, inserisce nella lista testo il numero
racchiuso fra parentesi tonde
411 char *c = strdup($1, 1);
412 if(strcmp(c, "-") == 0) {
413 c = (char *)malloc((strlen($1) + 3) * sizeof(char));
414 strcpy(c, "(");
415 strcat(c, $1);
416 strcat(c, ")");
417 put_testo(&Exp, c);
418 free(c);
419 } else
420 put_testo(&Exp, $1);
421 }
422 | T_DNUMBER {
423 if(!read) {
424 current_value = $1; current_type = "float";
425 if(array || index_element != NULL) {
426 dim++;
427 }
428 }
429
430 put_testo(&T, "float");
431 //Se è un numero negativo, inserisce nella lista testo il numero
racchiuso fra parentesi tonde
432 char *c = strdup($1, 1);
433 if(strcmp(c, "-") == 0) {
434 c = (char *)malloc((strlen($1) + 3) * sizeof(char));
435 strcpy(c, "(");
436 strcat(c, $1);
437 strcat(c, ")");
438 put_testo(&Exp, c);
439 free(c);
440 } else
441 put_testo(&Exp, $1);
442 }
443 | T_CONSTANT_ENCAPSED_STRING {
444 if(!read) {
445 current_value = $1; current_type = "char *";
446 if(array || index_element != NULL) {
447 dim++;
448 }
449 }
450
451 put_testo(&T, "char *");
452 put_testo(&Exp, $1);
453 }
454 | T_STRING {
455 current_type = isconstant($1, yylineno);
456 if(!read) {
457 current_value = $1;
458 if(array || index_element != NULL) {
459 dim++;
460 }
461 }
462
463 put_testo(&T, current_type);
464 put_testo(&Exp, $1);
465 }
466 ;
467
468 scalar:
469 common_scalar
470 |
471 |
472 ;
473
474 possible_comma:
475 /* empty */
476 |

```

```

477 ;
478
479 /* Tale regola definisce le variabile, gli elementi di un array, in sola lettura, o le costanti
utilizzate in una espressione o in una parte destra di una qualsiasi operazione di assegnazione,
semplice o complessa o in un costrutto. Per ogni elemento in sola lettura è effettuato un
controllo di esistenza nella Symbol table: se il controllo ha esito positivo, il nome
dell'elemento e il suo tipo sono inseriti rispettivamente nella lista T (Tipi) e Exp
(Espressione). */
480 expr:
481 r_variable { check($1, index_element, yylineno, true); }
482 | T_CONSTANT { check($1, 0, yylineno, true); }
483 | expr_without_variable
484 ;
485
486 r_variable:
487 variable { $$ = $1; read = true; }
488 ;
489
490 w_variable:
491 variable { $$ = $1; read = false; }
492 ;
493
494 variable:
495 T_VARIABLE { $$ = $1; }
496 | element_array
497 ;
498
499 element_array:
500 T_VARIABLE '[' T_LNUMBER ']' { $$ = $1; index_element = $3; }
501 | T_VARIABLE '[' T_VARIABLE ']' { $$ = $1; index_element = $3; }
502 ;
503
504 array_pair_list:
505 /* empty */
506 | non_empty_array_pair_list possible_comma
507 ;
508
509 non_empty_array_pair_list:
510 non_empty_array_pair_list ',' { put_testo(&Exp, ", "); } scalar
511 | scalar
512 ;
513
514 /* Tale regola definisce quali debbano essere gli elementi presenti nell'espressione
dell'istruzione PHP di stampa echo. Solamente nel caso di variabili, elementi di un array o di
costanti (si veda la regola encaps_var) viene richiamata la funzione echo_check (del file
symbol_table.h) che controlla l'esistenza, nella Symbol table o nella Constant table, nel caso
delle costanti, dei suddetti elementi. Se il controllo ha esito positivo, il nome dell'elemento
è inserito nella lista Exp (Espressione), mentre l'eventuale frase o stringa, nella lista
Phrase (Frase). */
515 encaps_list:
516 encaps_list encaps_var { echo_check($2, index_element, yylineno); }
517 | encaps_list T_STRING { strcat($2, " "); put_testo(&Phrase, $2); }
518 | encaps_list T_NUM_STRING { put_testo(&Phrase, $2); }
519 | encaps_list T_ENCAPSED_AND_WHITESPACE { put_testo(&Phrase, $2); }
520 /* empty */
521 ;
522
523 encaps_var:
524 T_VARIABLE { $$=$1; index_element = 0; }
525 | T_VARIABLE '[' T_NUM_STRING ']' { $$=$1; index_element = $3; }
526 | T_VARIABLE '[' T_VARIABLE ']' { $$=$1; index_element = $3; }
527 | T_CONSTANT { $$=$1; index_element = 0; }
528 ;
529
530
531 %%
532
533
534 /* La funzione pulizia, in caso di errore semantico fatale o per altri errori che comportino
l'interruzione della compilazione, chiude ed elimina il file di output f_out.c . */
535 void pulizia() {
536 chiudi_cancella_file(f_ptr);
537 }
538

```

```

539 /* La funzione yyerror stampa un messaggio di errore.
540 L'argomento è:
541 - s, la stringa contenente il messaggio di errore. */
542 void yyerror(char *s) {
543 _error++;
544 printf("\033[01;31mRiga %i. %s.\033[00m\n", yylineno, s);
545 }
546
547 /* La funzione main avvia il processo di compilazione e traduzione mediante la funzione interna
yyparse.
548 Gli argomenti sono:
549 - argc, il numero di argomenti passati da riga di comando;
550 - argv, i nomi dei file passati da riga di comando, contenenti il codice sorgente in PHP
da compilare e tradurre in C. */
551 main(int argc, char *argv[])
552 {
553 extern FILE *yyin;
554 ++argv; --argc;
555 int i;
556
557 for(i = 0; i < argc; i++) {
558 printf("Analisi del file %d: %s\n\n", i + 1, argv[i]);
559 if(fopen(argv[i], "r") != NULL)
560 yyin = fopen(argv[i], "r");
561 else {
562 printf("\033[01;31mERRORE: file %s non trovato.\033[00m\n", argv[i]);
563 exit(0);
564 }
565 //yydebug = 1;
566 yyparse();
567 printf("\n");
568 }
569
570 if(_error == 0 && _warning == 0) {
571 print_elements();
572 printf("\n\n\033[01;32mParsing riuscito.\033[00m\n");
573 } else if (_error == 0 && _warning != 0) {
574 print_elements();
575 printf("\n\n\033[01;33mParsing riuscito, ma sono presenti %i warnings.\033[00m\n", _warning);
576 } else {
577 printf("\n\n\033[01;31mParsing fallito:\033[00m");
578 if(_error > 1)
579 printf("\033[01;31m sono stati rilevati %i errori. \033[00m\n",
580 _error);
581 else
582 printf("\033[01;31m è stato rilevato %i errore. \033[00m\n", _error);
583 cancella_file();
584 }
585 }

```

# ***PHP2C***

## ***symbol\_table.h***

```

1 /*
2 +-----+
3 | PHP2C -- symbol_table.h |
4 +-----+
5 | Autori: BAVARO Gianvito |
6 | CAPURSO Domenico|
7 | DONVITO Marcello|
8 +-----+
9 */
10
11 /* Questo file contiene un insieme di funzioni utili alla gestione delle variabili, costanti ed
12 elementi di un array al fine di consentire una corretta applicazione delle regole semantiche. */
13 #include <stdlib.h> /* Serve per allocare memoria nella tabella dei simboli. */
14 #include <stdio.h> /* Serve per la gestione dei messaggi di errore semantico. */
15 #include <string.h> /* Serve per la gestione delle stringhe nella tabella dei simboli. */
16 #include "utility.h" /* Il file utility.h contiene delle funzioni e dei tipi di variabile di
17 supporto al processo di compilazione e traduzione. */
18 #include "gen_code.h" /* Il file gen_code.h contiene delle funzioni utili al processo di
19 traduzione. */
20 #include "uthash.h" /* Il file uthash.h è una libreria di gestione degli indici HASH,
21 utili per la gestione dei record della symbol table. */
22
23 #define NUM_CONSTANTS 3 /* Indica la dimensione della tabella delle costanti predefinite nel
24 linguaggio PHP. */
25 #define NUM_WARNINGS 6 /* Indica la dimensione dell'array warn contenente tutti i messaggi di
26 warning previsti. */
27
28 void pulizia(); /* Firma di una funzione definita nel file php_parser.h: quando si
29 verifica un errore semantico il file di traduzione verrà cancellato.
30 */
31 int notice = -1; /* Quando viene sollevato un warning, questo indice viene settato ad un
32 numero appropriato. */
33
34 /* Tale array contiene alcuni dei possibili messaggi di warning che il compilatore potrebbe
35 sollevare. L'accesso a uno specifico elemento è effettuato nel parser mediante l'indice notice.
36 */
37 char* warn[NUM_WARNINGS] = { "ATTENZIONE: l'uso di un operando di tipo stringa non è corretto
38 nel linguaggio target C.\n",
39 "ATTENZIONE: l'uso di un operando di tipo boolean non è corretto nel linguaggio target C.\n",
40 "ATTENZIONE: l'uso di un operando di tipo intero non è corretto nel linguaggio target C.\n",
41 "ATTENZIONE: l'uso di un operando di tipo float non è corretto nel linguaggio target C.\n",
42 "ATTENZIONE: l'uso di un elemento con offset negativo o superiore alla dimensione dell'array
43 potrebbe causare problemi nel linguaggio target C.\n",
44 "ATTENZIONE: la stampa di un elemento con offset negativo o superiore alla dimensione dell'array
45 potrebbe causare problemi nel linguaggio target C.\n" };
46
47 typedef struct CONSTANTS_TABLE /* Definizione della struttura della tabella delle costanti
48 predefinite del linguaggio PHP. */
49 {
50 char *ctM; /*parola maiuscola.
51 char *ctm; /*parola minuscola.
52 } CONSTANTS_TABLE;
53
54 /* La tabella delle costanti */
55 CONSTANTS_TABLE const_tab[NUM_CONSTANTS] = {
56 { "TRUE", "true" },
57 { "FALSE", "false" },
58 { "NULL", "null" }
59 };
60
61 typedef struct { /* Definizione della struttura della Symbol Table. */
62 char *nameToken; /*nome del token che si inserisce nella Symbol Table. Quesa stringa
63 viene usata come se fosse una chiave.*/
64 char *element; /*indica se è una "variable" o una "constant" o un "array".
65 char *type; /*il tipo primitivo: int, float, char * o bool.
66 char *value; /*il valore della variabile: pari a un valore, in caso di assegnazione
67 semplice, zero per un'assegnazione complessa o NULL se è un array.*/
68 int dim; /*dimensione dell'array.
69
70 UT_hash_handle hh; /* rende questa struttura indicizzabile mediante HASH.
71 } el_ST;
72
73 typedef el_ST *element_ptr; /* Puntatore all'elemento.

```



```

68 element_ptr table = NULL; // Puntatore alla tabella.
69
70 /* La funzione print_elements stampa tutti gli elementi contenuti nella Symbol table. */
71 void print_elements() {
72 element_ptr s;
73
74 printf("\n/* * * * * * * * SYMBOL TABLE * * * * * * */\n\n");
75 for(s = table; s != NULL; s = s->hh.next) {
76 printf("element name %s element %s type %s value %s dim %i\n", s->nameToken, s-
77 >element, s->type, s->value, s->dim);
78 }
79 printf("\n/* * * * * * * * * * * * * * * * * */\n\n");
80 }
81
82 /* La funzione find_element trova e restituisce un elemento della Symbol table. L'argomento è:
83 - nameToken, il nome del simbolo da cercare.
84 Restituisce l'elemento, se trovato, o NULL. */
85 element_ptr find_element(char *nameToken) {
86 element_ptr s;
87 HASH_FIND_STR(table, nameToken, s);
88
89 return s;
90 }
91
92 /* La funzione delete_element rimuove un elemento dalla Symbol table. L'argomento è:
93 - s, il puntatore all'elemento da rimuovere. */
94 void delete_element(element_ptr s) {
95 HASH_DEL(table, s);
96 }
97
98 /* La funzione add_element aggiunge un nuovo elemento nella Symbol table. Gli argomenti sono:
99 - nameToken, il nome del simbolo da aggiungere;
100 - element, il tipo di elemento ossia "constant", "variable" o "array";
101 - current_type, il tipo "int", "float", "char *" o "bool";
102 - current_value, il valore assegnato alla variabile: in particolare esso è zero se
103 l'assegnazione è complessa o NULL se è un array;
104 - dim, la dimensione dell'array, altrimenti verrà zero;
105 - nr, il numero riga segnalato dalla variabile yylineno di Flex. */
106 void add_element(char *nameToken, char *element, char *current_type, char *current_value, int
107 dim, int nr) {
108 element_ptr s;
109 element_ptr exist = find_element(nameToken);
110
111 //se non esiste inserisce l'elemento nella Symbol table.
112 if(!exist) {
113 s = malloc(sizeof(el_ST));
114 s->nameToken = nameToken;
115 s->element = element;
116 s->type = current_type;
117 s->value = current_value;
118 s->dim = dim;
119 HASH_ADD_KEYPTR(hh, table, s->nameToken, strlen(s->nameToken), s);
120 /*se si tenta di inserire una costante viene sollevato un errore fatale, che blocca la
121 compilazione. Si sta cercando di ridefinire una costante.*/
122 } else if (strcmp(exist->element, "constant") == 0) {
123 printf("\033[01;31mRiga %i. [FATALE] ERRORE SEMANTICO: ridefinizione di una
124 costante.\033[00m\n", nr);
125 printf("\n\n\033[01;31mParsing fallito.\033[00m\n");
126 pulizia();
127 exit(0);
128 }
129 //altrimenti è una riassegnazione di valori a una variabile: occorre verificare che la
130 riassegnazione non violi il tipo precedentemente definito.
131 } else {
132 //in caso di non violazione verrà aggiornato il valore della
133 variabile.
134 if(strcmp(exist->type, current_type) == 0) {
135 delete_element(exist);
136 exist->value = current_value;
137 HASH_ADD_KEYPTR(hh, table, exist->nameToken, strlen(exist-
138 >nameToken), exist);
139 //altrimenti sarà lanciato un errore semantico fatale.
140 } else {
141 printf("\033[01;31mRiga %i. [FATALE] ERRORE SEMANTICO: l'assegnazione
142 viola il tipo primitivo della variabile.\033[00m\n", nr);

```

```

135 printf("\n\n033[01;31mParsing fallito.\033[00m\n");
136 pulizia();
137 exit(0);
138 }
139 }
140 }
141
142 /* La funzione type_checking controlla se le varie operazioni matematiche di +, -, *, /, %, ++,
-- , <, <=, >, >= abbiano come operandi variabili intere (int) o reali (float). In caso di
utilizzo di variabili booleane o di stringhe vengono generati dei warning mediante
l'assegnazione di un opportuno valore alla variabile notice.
Gli argomenti sono:
143 - TT, una lista definita in utility.h che contiene tutti i tipi associati agli operandi;
144 - nr, il numero riga segnalato dalla variabile yylineno di Flex.
145 Restituisce il tipo da assegnare alla variabile da aggiungere o aggiornare (solo il valore)
146 nella Symbol table. */
147 char *type_checking(testo *TT, int nr)
148 {
149 testo *punt = TT;
150 char *tipo;
151 char *current_type = "int";
152
153 //get_testo(TT, "TIPE");
154
155 while(punt != NULL) {
156
157 if(strcmp(punt->tes, "float") == 0) {
158 current_type = "float";
159 }
160
161 if(strcmp(punt->tes, "char *") == 0) {
162 notice = 0;
163 //break;
164 }
165 if(strcmp(punt->tes, "bool") == 0) {
166 notice = 1;
167 //break;
168 }
169 punt = punt->next;
170 }
171
172 return current_type;
173 }
174
175 /* La funzione type_array_checking controlla se:
176 - le varie operazioni di creazione di un array siano di tipo omogeneo;
177 - l'assegnazione a un elemento dell'array abbia come operandi variabili omogenee e di
178 tipo identico a quello dell'array dichiarato nella Symbol table.
179 Gli argomenti sono:
180 - TT, una lista definita in utility.h che contiene tutti i tipi associati agli operandi;
181 - context, indica il contesto di utilizzo di tale funzione:
182 -> 'c' indica "create" ovvero la creazione di un array. La funzione con tale
183 contesto è richiamata da un'azione semantica di una regola del parser;
184 -> 's' indica "single" ossia quando è in atto un'assegnazione di un solo valore
185 a un elemento di array;
186 -> 'm' indica "multiple" ossia quando è in atto un'assegnazione multipla a un
187 elemento di un array.
188 Le funzioni con contesto 's' e 'm' sono richiamate dalla funzione check_element
189 spiegata più avanti;
190 - exist, l'elemento ricercato precedentemente nella funzione check_element (è l'array di
191 appartenenza dell'elemento);
192 - nr, il numero riga segnalato dalla variabile yylineno di Flex. */
193 void type_array_checking(testo *TT, char context, element_ptr exist, int nr) {
194 testo *punt = TT;
195 char *tipo;
196
197 //get_testo(TT, "TIPI");
198
199 switch(context) {
200 //Controlla la creazione degli array: omogeneità dei tipi
201 case 'c':
202 if(TT != NULL)
203 tipo = punt->tes;
204 while(TT != NULL) {

```

```

205 if(strcmp(TT->tes, tipo) != 0)
206 {
207 printf("\033[01;31mRiga %i. [FATALE] ERRORE
SEMANTICO: l'uso di un operando di tipo non omogeneo in un array tipizzato non è corretto nel
linguaggio target C.\033[00m\n", nr);
208 printf("\n\n\033[01;31mParsing fallito.\033[00m\n");
209 pulizia();
210 exit(0);
211 }
212 TT = TT->next;
213 }
214 //get_testo(TT);
215 break;
216 //assegnazione singola
217 case 's':
218 if(strcmp(exist->type, TT->tes) != 0) {
219 printf("\033[01;31mRiga %i. [FATALE] ERRORE SEMANTICO:
l'assegnazione viola l'omogeneità degli elementi dell'array \"%s\".\033[00m\n", nr, exist-
>nameToken);
220 printf("\n\n\033[01;31mParsing fallito.\033[00m\n");
221 pulizia();
222 exit(0);
223 }
224 break;
225 //assegnazione multipla (espressioni)
226 case 'm':
227 tipo = exist->type;
228 while(TT != NULL) {
229 if(strcmp(TT->tes, tipo) != 0)
230 {
231 printf("\033[01;31mRiga %i. [FATALE] ERRORE
SEMANTICO: l'assegnazione viola l'omogeneità degli elementi dell'array \"%s\".\033[00m\n", nr,
exist->nameToken);
232 printf("\n\n\033[01;31mParsing fallito.\033[00m\n");
233 pulizia();
234 exit(0);
235 }
236 TT = TT->next;
237 }
238 break;
239 }
240 }
241
242 /* La funzione check_element_gen_code controlla se l'elemento indicato esiste o meno nella
Symbol table. Tale funzione è utilizzata nella traduzione delle istruzioni di assegnazione
(funzioni gen_create_array e gen_assignment dichiarate nel file gen_code.h) :
243 - se si assegna un valore a una variabile nuova e, quindi non presente in tabella, si
riporta anche il tipo associato;
244 - altrimenti se si riassegna un valore a una variabile già esistente in tabella, non
si riporta il tipo associato.
245 L'unico argomento è:
246 - nameToken, il nome del simbolo da cercare.
247 Restituisce 1 se l'elemento è stato trovato altrimenti 0. */
248 int check_element_gen_code(char *nameToken) {
249 if(find_element(nameToken) != NULL)
250 return 1;
251
252 return 0;
253 }
254
255 /* La funzione check_index è richiamata in caso di assegnazione di un valore a un elemento di un
array (in fase di scrittura, quindi) al fine di controllare se l'elemento, e il suo indice (o
offset), siano validi. Essa controlla se l'elemento indicato esista nella Symbol table e, in
caso affermativo, effettua dei controlli sull'offset:
256 -> se è un numero, converte il contenuto della stringa nel corrispondente valore intero;
257 -> se è una variabile si accerta della sua esistenza e, nel caso esista, converte il
contenuto della stringa nel corrispondente valore intero;
258 -> un ultimo controllo è dedicato al valore intero dell'offset. Se è minore di zero o
maggiore della dimensione massima dichiarata nella Symbol table, è lanciato un
warning.
259 Gli argomenti sono:
260 - nameToken, il nome dell'elemento dell'array;
261 - offset, l'indice dell'elemento;
262 - nr, il numero riga segnalato dalla variabile yylineno di Flex. */

```

```

266 void check_index(char *nameToken, char *offset, int nr) {
267 int index;
268 element_ptr exist = find_element(nameToken);
269 element_ptr exist_index;
270 //se l'elemento non esiste lancia un errore semantico fatale.
271 if(!exist) {
272 printf("\033[01;31m\033[01;31mRiga %i. [FATALE] ERRORE SEMANTICO: variabile
273 \"%s\" non definita.\033[00m\n", nr, nameToken);
274 printf("\n\n\033[01;31mParsing fallito.\033[00m\n");
275 pulizia();
276 exit(0);
277 } else {
278 //mediante la funzione isnumeric contenuta nel file utility.h ci si accerta che
279 l'offset sia un numero
280 if(isnumeric(offset)) {
281 //in caso affermativo la funzione atoi converte il contenuto della
282 stringa (in tal caso un numero) nel corrispondente valore
283 intero.
284 index = atoi
285 (offset);
286 } else {
287 //in caso contrario l'offset è una variabile. Occorre controllare che
288 esista mediante la funzione find_element.
289 exist_index = find_element(offset);
290 //se esiste
291 if(exist_index) {
292 //controlla che un indice non sia intero poichè è ammissibile,
293 quindi viene effettuato un controllo sulla variabile offset. Se il tipo è diverso da "int" viene
294 visualizzato un errore semantico fatale.
295 if(strcmp(exist_index->type, "int") != 0) {
296 printf("\033[01;31mRiga %i. [FATALE] ERRORE
297 SEMANTICO: l'uso di un elemento con offset non intero non è ammissibile.\033[00m\n",
298 nr);
299 printf("\n\n\033[01;31mParsing fallito.\033[00m\n");
300 pulizia();
301 exit(0);
302 }
303 //la funzione atoi converte il valore della variabile nel
304 corrispondente valore intero.
305 index = atoi(exist_index->value);
306 } else {
307 //se non esiste viene visualizzato un errore semantico fatale.
308 printf("\033[01;31mRiga %i. [FATALE] ERRORE SEMANTICO:
309 variabile \"%s\" non definita.\033[00m\n", nr, offset);
310 printf("\n\n\033[01;31mParsing fallito.\033[00m\n");
311 pulizia();
312 exit(0);
313 }
314 }
315 //se l'indice è minore di 0 o maggiore della dimensione specificata nella Symbol
316 table viene visualizzato un messaggio di warning, assegnano alla variabile notice un valore
317 appropriato.
318 if(index < 0 || index >= exist->dim) {
319 notice = 4;
320 }
321 }
322 }
323
324 /* La funzione check_element è richiamata per controllare l'esistenza delle variabili o degli
325 elementi di un array, in sola lettura, o delle costanti, ossia tutte quelle variabili
326 utilizzate nelle operazioni:
327 - composizione di un'espressione, anche complessa, di assegnazione;
328 - costrutti tipo, if, for (condizione), switch, while e do-while;
329 - operatori ++ o -- (solo per le variabili).
330 Tale funzione ha molte cose in comune con la precedente, ma viene richiamata in molti punti
331 differenti della grammatica:
332 -> nel caso di lettura di una variabile, di un elemento di un array o di una costante,
333 operando di un'espressione o di un costrutto, la funzione è richiamata con valore
334 read = true. Dopo il controllo positivo nella Symbol table, il tipo associato verrà
335 inserito nella lista T (Tipi).
336 -> nel caso delle assegnazioni di espressioni a un elemento di un array, essa è
337 richiamata dopo la creazione dell'espressione di assegnazione con valore read = false.
338 Poichè l'espressione è stata generata avremo a disposizione la lista T, e in base al

```

```

326 numero di elementi presenti verrà chiamata la funzione type_array_checking nel
contesto
327 's' o 'm' per un controllo sull'omogeneità dei tipi.
328 Gli argomento sono:
329 - nameToken, il nome della variabile o dell'elemento di un array;
330 - offset, l'indice dell'elemento dell'array;
331 - nr, il numero riga segnalato dalla variabile yylineno di Flex;
332 - read, specifica se l'elemento è analizzato in lettura o scrittura (solo per elementi
di
333 un array).
334 Restituisce l'elemento identificato dal nameToken, trovato nella Symbol table. */
335 element_ptr check_element(char *nameToken, char *offset, int nr, bool read)
336 {
337 int index;
338 char *tipo;
339 char *el_array;
340
341 element_ptr exist = find_element(nameToken);
342 element_ptr exist_index;
343 //se l'elemento non esiste lancia un errore semantico fatale.
344 if(!exist) {
345 printf("\033[01;31m\033[01;31mRiga %i. [FATALE] ERRORE SEMANTICO: variabile
\"%s\" non definita.\033[00m\n", nr, nameToken);
346 printf("\n\n\033[01;31mParsing fallito.\033[00m\n");
347 pulizia();
348 exit(0);
349
350 } else {
351 //se è una variabile di sola lettura o una costante lo aggiungo alle liste T ed
Exp (Espressione).
352 if(((strcmp(exist->element, "variable") == 0) || (strcmp(exist->element,
"constant") == 0)) && read) {
353 //aggiungo il tipo dell'elemento alla lista T.
354 put_testo(&T, exist->type);
355 //aggiungo il nome dell'elemento alla lista Exp.
356 put_testo(&Exp, exist->nameToken);
357 } else {
358 //se è un elemento di un array di sola lettura (non è un assegnazione a
se stesso) lo aggiungo alle liste T ed Exp (Espressione).
359 if(read) {
360 //aggiungo il tipo dell'elemento alla lista T.
361 put_testo(&T, exist->type);
362 //viene legato al nome dell'array (identificato dal nameToken)
l'offset: nome[offset]. Successivamente aggiungo il nome composto alla lista Exp.
363 el_array = nameToken;
364 strcat(el_array, "[");
365 strcat(el_array, offset);
366 strcat(el_array, "]");
367 strcat(el_array, "\0");
368 put_testo(&Exp, el_array);
369 }
370 //mediante la funzione isnumeric contenuta nel file utility.h ci si
accerta che l'offset sia un numero
371 if(isnumeric(offset)) {
372 //in caso affermativo la funzione atoi converte il contenuto
della stringa (in tal caso un numero) nel corrispondente valore intero.
373 index = atoi(offset);
374 } else {
375 //in caso contrario l'offset è una variabile. Occorre
controllare che esista mediante la funzione find_element.
376 exist_index = find_element(offset);
377
378 //se esiste.
379 if(exist_index) {
380 //controlla che un indice non sia intero poichè è
ammissibile, quindi viene effettuato un controllo sulla variabile offset. Se il tipo è diverso
da "int" viene visualizzato un errore semantico fatale.
381 if(strcmp(exist_index->type, "int") != 0) {
382 printf("\033[01;31mRiga %i. [FATALE] ERRORE
SEMANTICO: l'uso di un elemento con offset non intero non è ammissibile.\033[00m\n", nr);
383 printf("\n\n\033[01;31mParsing fallito.\033[00m
\n");
384 pulizia();
385 exit(0);

```

```

386 }
387 //la funzione atoi converte il valore della variabile
nel corrispondente valore intero.
388 index = atoi(exist_index->value);
389 } else {
390 //se non esiste viene visualizzato un errore semantico
fatale.
391 printf("\033[01;31mRiga %i. [FATALE] ERRORE
SEMANTICO: variabile \"%s\" non definita.\033[00m\n", nr, offset);
392 printf("\n\n\033[01;31mParsing fallito.\033[00m\n");
393 pulizia();
394 exit(0);
395 }
396 }
397
398 //se l'indice è minore di 0 o maggiore della dimensione specificata
nella Symbol table viene visualizzato un messaggio di warning, assegnando alla variabile notice
un valore appropriato.
399 if(index < 0 || index >= exist->dim) {
400 notice = 4;
401 }
402
403 //get_testo(T, "TIPI");
404
405 //se la lista T non è vuota e se sto assegnando un valore all'elemento
di un array (scrittura, read = false), sulla base del numero di elementi nella lista T
effettuo un controllo sull'omogeneità dei tipi, secondo il contesto 's' o 'm'.
406 if(countelements(T) != 0 && !read) {
407
408 switch(countelements(T)) {
409 case 1: type_array_checking(T, 's', exist, nr);
410 break;
411
412 default: type_array_checking(T, 'm', exist, nr);
413 break;
414 }
415 }
416 }
417 }
418
419 return exist;
420 }
421
422 /* La funzione echo_check verifica se le variabili, gli elementi di un array o le costanti
423 specificate nell'istruzione echo esistono o meno nella Symbol table. Nel caso degli elementi
424 di un array viene controllato l'offset secondo le stesse modalità descritte in precedenza.
425 Sulla base del tipo di variabile analizzata vengono assegnate alle liste Exp e Phrase
426 (definite nel file utility.h) rispettivamente gli elementi e la frase che saranno
427 successivamente tradotti nella funzione C printf mediante la funzione gen_echo (contenuta
428 nel file gen_code.h).
429 Gli argomenti sono:
430 - nameToken, il nome della variabile, dell'elemento di un array o di una costante
431 contenuta nell'espressione associata all'istruzione echo;
432 - offset, l'indice dell'elemento dell'array;
433 - nr, il numero riga segnalato dalla variabile yylineno di Flex. */
434 void echo_check(char *nameToken, char *offset, int nr)
435 {
436 int index;
437 char *tmp = ") ? \"true\" : \"false\""; //variabile utilizzata per gestire la
traduzione della stampa di valori booleani.
438 char *el_array;
439 element_ptr exist = find_element(nameToken);
440 element_ptr exist_index;
441 //se l'elemento non esiste lancia un errore semantico fatale.
442 if(!exist) {
443 printf("\033[01;31m\033[01;31mRiga %i. [FATALE] ERRORE SEMANTICO: variabile
\"%s\" non definita.\033[00m\n", nr, nameToken);
444 printf("\n\n\033[01;31mParsing fallito.\033[00m\n");
445 pulizia();
446 exit(0);
447 } else {
448 //se l'elemento esiste e se è un array vengono effettuati gli stessi controlli
sull'offset.
449 if(strcmp(exist->element, "array") == 0) {

```

```

450 if(offset == NULL) {
451 printf("\033[01;31m\033[01;31mRiga %i. [FATALE] ERRORE
SEMANTICO: offset non definito.\033[00m\n", nr);
452 printf("\n\n\033[01;31mParsing fallito.\033[00m\n");
453 pulizia();
454 exit(0);
455 }
456 if(isnumeric(offset)) {
457 index = atoi(offset);
458 } else {
459 exist_index = find_element(offset);
460
461 //se esiste.
462 if(exist_index) {
463 //controlla che un indice non sia intero poichè è
inammissibile. Quindi viene effettuato un controllo sulla variabile offset. Se il tipo è
diverso da "int" viene visualizzato un errore semantico fatale.
464 if(strcmp(exist_index->type, "int") != 0) {
465 printf("\033[01;31mRiga %i. [FATALE] ERRORE
SEMANTICO: l'uso di un elemento con offset non intero non è ammissibile.\033[00m\n", nr);
466 printf("\n\n\033[01;31mParsing fallito.\033[00m
\n");
467 pulizia();
468 exit(0);
469 }
470 //la funzione atoi converte il valore della variabile
nel corrispondente valore intero.
471 index = atoi(exist_index->value);
472 } else {
473 printf("\033[01;31m\033[01;31mRiga %i. [FATALE]
ERRORE SEMANTICO: variabile \"%s\" non definita.\033[00m\n", nr, nameToken);
474 printf("\n\n\033[01;31mParsing fallito.\033[00m\n");
475 pulizia();
476 exit(0);
477 }
478 }
479 //l'offset viene legato al nome dell'array (identificato dal
nameToken): nome[offset].
480 el_array = nameToken;
481 strcat(el_array, "[");
482 strcat(el_array, offset);
483 strcat(el_array, "]");
484 strcat(el_array, "\0");
485 //se il tipo dell'elemento è booleano viene aggiunta alla lista Exp la
variabile tmp. In entrambi i casi è aggiunta alla lista Exp l'elemento dell'array.
486 if (strcmp(exist->type, "bool") == 0) {
487 char *c = (char *)malloc((strlen(el_array) + strlen(tmp)
+ 1) * sizeof(char));
488 strcpy(c, "(");
489 strcat(c, el_array);
490 strcat(c, tmp);
491 put_testo(&Exp, c);
492 free(c);
493 } else
494 put_testo(&Exp, el_array);
495
496 if(index < 0 || index >= exist->dim) {
497 notice = 5;
498 }
499 } else {
500 //nel caso in cui l'elemento sia una variabile o una costante: se il
tipo dell'elemento è booleano viene aggiunta alla lista Exp la variabile tmp. In entrambi i casi
è aggiunta alla lista Exp la variabile o la costante.
501 if (strcmp(exist->type, "bool") == 0) {
502 char *c = (char *)malloc((strlen(nameToken) + strlen
(tmp) + 1) * sizeof(char));
503 strcpy(c, "(");
504 strcat(c, nameToken);
505 strcat(c, tmp);
506 put_testo(&Exp, c);
507 free(c);
508 } else
509 put_testo(&Exp, nameToken);
510 }

```

```

511
512 //in base al tipo dell'elemento viene aggiunto alla lista testo il giusto
513 identificatore di variabili utilizzato dal C nella funzione printf.
514 if(strcmp(exist->type, "int") == 0)
515 put_testo(&Phrase, " %i ");
516 else if(strcmp(exist->type, "float") == 0)
517 put_testo(&Phrase, " %f ");
518 else
519 put_testo(&Phrase, " %s ");
520 }
521 }
522
523 /* La funzione isconstant verifica se la stringa letta (identificata dal token T_STRING) sia
524 una costante predefinita del PHP o se sia una costante definita dall'utente mediante la
525 funzione define.
526 Gli argomenti sono:
527 - string, la stringa da controllare;
528 - nr, il numero riga segnalato dalla variabile yylineno di Flex.
529 Restituisce il tipo della costante per effettuare i controlli di tipo (type_checking). */
530 char *isconstant(char *string, int nr)
531 {
532 int i;
533 int trov = 0;
534 char *current_type = "bool";
535 // 1° controllo nella tabella delle costanti predefinite
536 for(i = 0; i < NUM_CONSTANTS; i++)
537 {
538 if(strcmp(string, const_tab[i].ctM) == 0)
539 trov = 1;
540 if(strcmp(string, const_tab[i].ctm) == 0)
541 trov = 1;
542 }
543 //solo se non è una costante predefinita effettuo un ulteriore controllo.
544 if(trov == 0) {
545 // 2° controllo nella symbol table.
546 element_ptr exist = find_element(string);
547
548 if(exist && (strcmp(exist->element, "constant") == 0)) {
549 current_type = exist->type;
550 trov = 1;
551 }
552 }
553 //se non è una costante viene lanciato un errore semantico fatale.
554 if(trov == 0) {
555 printf("\033[01;31mRiga %i. [FATALE] ERRORE SEMANTICO: stringa \"%s\" non
556 riconosciuta.\033[00m\r\n", nr, string);
557 printf("\n\n\033[01;31mParsing fallito.\033[00m\n");
558 pulizia();
559 exit(0);
560 }
561 return current_type;
562 }

```



# ***PHP2C***

## ***gen\_code.h***

```

1 /*
2 +-----+
3 | PHP2C -- gen_code.h |
4 +-----+
5 | Autori: BAVARO Gianvito |
6 | CAPURSO Domenico |
7 | DONVITO Marcello |
8 +-----+
9 */
10
11 /* Questo file contiene un insieme di funzioni utili alla traduzione in C di alcuni costrutti e
istruzione in PHP. Tuttavia la maggior parte della traduzione viene svolta mediante funzioni
direttamente inserite nella definizione della grammatica, contenuta nel file php_parser.y. */
12
13 #include <stdio.h> /* Serve per la gestione dei messaggi di
14 errore semantico. */
15 #include <string.h> // Serve per la gestione delle stringhe
16
17 #define PATH "/home/utente/traduttore/f_out.c" /* Rappresenta il percorso dove verrà
18 salvato il file "f_out.c" contenente
19 la traduzione in C. */
20
21 char *op_name[] = { "=", "+=", "-=", "*=", "/=", "%=" };/* Un array contenente tutti i possibili
22 operatori di assegnazione. */
23
24 /* La funzione apri_file apre il file f_out.c che conterrà il codice intermedio C, frutto della
compilazione e traduzione.
25 Restituisce il puntatore al file. */
26 FILE *apri_file() {
27 FILE *f_ptr;
28
29 if ((f_ptr = fopen(PATH, "w")) == NULL) {
30 printf("\033[01;31mERRORE: apertura del file f_out.c fallita. Directory %s non
esistente.\033[00m\n", PATH);
31 exit(0);
32 }
33
34 return f_ptr;
35 }
36
37 /* La funzione chiudi_file chiude il file f_out.c precedentemente aperto. */
38 void chiudi_file(FILE *f_ptr) {
39 //forza l'effettiva scrittura di eventuali dati presenti nel buffer, prima della
chiusura.
40 fflush(f_ptr);
41 //chiude il file.
42 fclose(f_ptr);
43 }
44
45 /* La funzione chiudi_cancella_file, in caso di errore semantico o grave, chiude ed elimina il
file f_out.c .
46 L'argomento è:
47 - f_ptr, il puntatore al file da chiudere. */
48 void chiudi_cancella_file(FILE *f_ptr) {
49 if(f_ptr != NULL)
50 chiudi_file(f_ptr);
51
52 if (remove(PATH) == -1)
53 printf("\033[01;31mERRORE: impossibile cancellare il file.\033[00m\n");
54 }
55
56 /* La funzione cancella_file elimina il file f_out.c . Essa è richiamata ad ogni inizio
compilazione, in modo tale da poter ricreare un nuovo file. */
57 void cancella_file() {
58 FILE *f_ptr;
59 //solo se il file esiste sarà cancellato.
60 if ((f_ptr = fopen(PATH, "r"))) {
61 fclose(f_ptr);
62 if (remove(PATH) == -1)
63 printf("\033[01;31mERRORE: impossibile cancellare il file.\033[00m\n");
64 }
65 }
66
67 /* La funzione gen_header genera l'header del file C, scrivendo due istruzioni di default stdio

```

```

e string, il metodo main e la definizione del tipo boolean.
68 L'argomento è:
69 - f_ptr, il puntatore al file nel quale stampare la traduzione. */
70 void gen_header(FILE* f_ptr) {
71 fprintf(f_ptr, "#include <stdio.h>\n");
72 fprintf(f_ptr, "#include <string.h>\n\n");
73 fprintf(f_ptr, "void main(void) {\n");
74 fprintf(f_ptr, "\\typedef enum { false, true } bool;\n");
75 }
76
77 /* La funzione gen_constant genera le costanti dichiarate.
78 Gli argomenti sono:
79 - f_ptr, il puntatore al file nel quale stampare la traduzione;
80 - nameConstant, il nome della costante;
81 - type, il tipo della costante;
82 - value, il valore della costante. */
83 void gen_constant(FILE* f_ptr, char *nameConstant, char *type, char *value) {
84 //se è una costante di tipo stringa allora devo stampare un array di caratteri.
85 if(strcmp(type, "char *") == 0)
86 fprintf(f_ptr, "const char %s[] = %s;\n", nameConstant, value);
87 else
88 fprintf(f_ptr, "const %s %s = %s;\n", type, nameConstant, value);
89 }
90
91 /* La funzione gen_expression genera le espressioni, create da composizioni di variabili,
92 elementi di array e costanti legate fra loro dagli operatori previsti dalla grammatica. Tutti
93 questi elementi sono stati precedentemente inseriti nella lista Exp.
94 L'argomento è:
95 - Exp, la lista contenenti gli elementi delle espressioni.
96 Restituisce l'espressione. */
97 char *gen_expression(testo *Exp) {
98 char *expression = NULL;
99
100 //get_testo(Exp, "ESPR");
101 //assegna a expression il primo valore della lista Exp.
102 if(Exp != NULL) {
103 expression = Exp->tes;
104 Exp = Exp->next;
105 }
106 //concatena tutti i successivi elementi della lista.
107 while(Exp != NULL) {
108 strcat(expression, Exp->tes);
109 Exp = Exp->next;
110 }
111 if(Exp != NULL) {
112 strcat(expression, "\\0");
113 }
114 return expression;
115 }
116
117 /* La funzione gen_echo_expression è simile alla precedente ma finalizzata alla creazione di
118 espressioni per l'istruzione echo.
119 Gli argomenti sono:
120 - f_ptr, il puntatore al file nel quale stampare la traduzione;
121 - Exp, la lista contenenti gli elementi delle espressioni.
122 Restituisce l'espressione. */
123 char *gen_echo_expression(FILE* f_ptr, testo *Exp) {
124 char *expression = NULL;
125 //se gli elementi sono in numero maggiore di uno è prevista la stampa della ",".
126 int elements = countelements(Exp);
127
128 //get_testo(Exp, "GEN ECHO ESPR");
129
130 //assegna a expression il primo valore della lista Exp.
131 if(Exp != NULL) {
132 expression = Exp->tes;
133 Exp = Exp->next;
134 }
135 //concatena tutti i successivi elementi della lista.
136 while(Exp != NULL) {
137 if(elements > 1)
138 strcat(expression, ", ");
139 strcat(expression, Exp->tes);

```

```

138 Exp = Exp->next;
139 }
140 if(Exp != NULL) {
141 strcat(expression, "\0");
142 }
143
144 return expression;
145 }
146
147 /* La funzione print_expression genera le espressioni che stamperà nel file f_out.c, create da
composizioni di variabili, elementi di array e costanti legate fra loro dagli operatori previsti
dalla grammatica. Tutti questi elementi sono stati precedentemente inseriti nella lista Exp.
148 Gli argomenti sono:
149 - f_ptr, il puntatore al file nel quale stampare la traduzione;
150 - Exp, la lista contenenti gli elementi delle espressioni. */
151 void print_expression(FILE* f_ptr, testo *Exp) {
152 char *expression;
153
154 //get_testo(Exp, "ESPR");
155
156 expression = Exp->tes;
157 Exp = Exp->next;
158
159 while(Exp != NULL) {
160 strcat(expression, Exp->tes);
161 Exp = Exp->next;
162 }
163 strcat(expression, "\0");
164 fprintf(f_ptr, "%s", expression);
165 }
166
167 /* La funzione gen_create_array stampa la dichiarazione di un array.
168 Gli argomenti sono:
169 - f_ptr, il puntatore al file nel quale stampare la traduzione;
170 - name_array, il nome dell'array;
171 - type, il tipo dell'array;
172 - Exp, la lista contenenti gli elementi delle espressioni. */
173 void gen_create_array(FILE* f_ptr, char *name_array, char *type, testo *Exp) {
174 char *expression = gen_expression(Exp);
175 //Se l'array non è stato definito viene stampato anche il tipo, altrimenti no.
176 if (check_element_gen_code(name_array))
177 fprintf(f_ptr, "%s[] = { %s }", name_array, expression);
178 else
179 fprintf(f_ptr, "%s %s[] = { %s }", type, name_array, expression);
180 }
181
182
183 /* La funzione gen_assignment stampa le dichiarazioni di una variabile o di un elemento di un
array.
184 Gli argomenti sono:
185 - f_ptr, il puntatore al file nel quale stampare la traduzione;
186 - index, indice per identificare il tipo di operatore di assegnazione contenuto
nell'array op_name;
187 - left_var, il nome della variabile o dell'elemento dell'array;
188 - type, il tipo dell'array;
189 - index_element, l'offset dell'elemento di un array;
190 - Exp, la lista contenenti gli elementi delle espressioni. */
191 void gen_assignment(FILE* f_ptr, int index, char *left_var, char *type, char *index_element,
testo *Exp, bool array) {
192 char *expression = gen_expression(Exp);
193
194 //Se è una variabile
195 if(!array) {
196 //Se la variabile è stata definita viene stampato anche il tipo, altrimenti no.
197 if (check_element_gen_code(left_var))
198 fprintf(f_ptr, "%s %s %s", left_var, op_name[index], expression);
199 else
200 fprintf(f_ptr, "%s %s %s %s", type, left_var, op_name[index],
expression);
201 } else {
202 fprintf(f_ptr, " %s %s", op_name[index], expression);
203 }
204 }
205
206

```

```

207 /* La funzione gen_if genera l'istruzione condizionale if.
208 Gli argomenti sono:
209 - f_ptr, il puntatore al file nel quale stampare la traduzione;
210 - Exp, la lista contenenti gli elementi delle espressioni. */
211 void gen_if(FILE* f_ptr, testo *Exp) {
212 char *expression = gen_expression(Exp);
213 fprintf(f_ptr, "if(%s) {\n", expression);
214 }
215
216 /* La funzione gen_elseif genera l'istruzione condizionale else if.
217 Gli argomenti sono:
218 - f_ptr, il puntatore al file nel quale stampare la traduzione;
219 - Exp, la lista contenenti gli elementi delle espressioni. */
220 void gen_elseif(FILE* f_ptr, testo *Exp) {
221 char *expression = gen_expression(Exp);
222 fprintf(f_ptr, " else if(%s) {\n", expression);
223 }
224
225 /* La funzione gen_while genera l'istruzione di ciclo while.
226 Gli argomenti sono:
227 - f_ptr, il puntatore al file nel quale stampare la traduzione;
228 - Exp, la lista contenenti gli elementi delle espressioni. */
229 void gen_while(FILE* f_ptr, testo *Exp) {
230 char *expression = gen_expression(Exp);
231 fprintf(f_ptr, "while(%s) {\n", expression);
232 }
233
234 /* La funzione gen_switch genera il costrutto switch.
235 Gli argomenti sono:
236 - f_ptr, il puntatore al file nel quale stampare la traduzione;
237 - Exp, la lista contenenti gli elementi delle espressioni. */
238 void gen_switch(FILE* f_ptr, testo *Exp) {
239 char *expression = gen_expression(Exp);
240 fprintf(f_ptr, "switch(%s) {\n", expression);
241 }
242
243 /* La funzione gen_echo genera l'istruzione di stampa a video printf.
244 Gli argomenti sono:
245 - f_ptr, il puntatore al file nel quale stampare la traduzione;
246 - Exp, la lista contenenti gli elementi delle espressioni;
247 - Phrase, la lista contenenti gli elementi della frase da associare all'espressione. */
248 void gen_echo(FILE* f_ptr, testo *Exp, testo *Phrase) {
249 char *phrase = gen_expression(Phrase);
250 char *expression = gen_echo_expression(f_ptr, Exp);
251
252 if(phrase != NULL && expression != NULL)
253 fprintf(f_ptr, "printf(\"%s\\\", %s);\n", phrase, expression);
254 else if(phrase != NULL)
255 fprintf(f_ptr, "printf(\"%s\\\");\n", phrase);
256 else
257 fprintf(f_ptr, "printf(%s);\n", expression);
258 }
259
260 /* La funzione gen_tab genera il simbolo tab "\t".
261 Gli argomenti sono:
262 - f_ptr, il puntatore al file nel quale stampare il simbolo;
263 - ntab, il numero di stampa del simbolo tab. */
264 void gen_tab(FILE *f_ptr, int ntab) {
265 int i;
266 for(i = 0; i < ntab; i++)
267 fprintf(f_ptr, "\t");
268 }

```

***PHP2C***

***utility.h***

```

1 /*
2 +-----+
3 | PHP2C -- utility.h |
4 +-----+
5 | Autori: BAVARO Gianvito |
6 | CAPURSO Domenico |
7 | DONVITO Marcello |
8 +-----+
9 */
10
11 /* Questo file contiene un insieme di strutture dati e funzioni di utilità e supporto per il
12 processo di compilazione e traduzione. */
13
14 typedef enum { false, true } bool; /* Struttura per un nuovo tipo di dato: bool.
15 Rappresenta i valori booleani. */
16
17 typedef struct testo /* Struttura per un nuovo tipo di dato: testo.
18 E' una lista concatenata di stringhe. */
19 {
20 char *tes;
21 struct testo *next;
22 } testo;
23
24 testo *T; /* Lista concatenata contenente i tipi delle variabili, elementi
25 di array o costanti. Utile per il type_checking.*/
26
27 testo *Exp; /* Lista concatenata contenente gli elementi di una espressione
28 (variabili, elementi di array, costanti, operatori). Utile per
29 stampare intere espressioni nel file f_out.c contenente la
30 traduzione in C. */
31
32 testo *Phrase; /* Lista concatenata contenente frasi o parole (non keywords).
33 Utile per stampare intere frasi, anche associate a
34 espressioni, nel file f_out.c . */
35
36 /* La funzione clear cancella il contenuto di ogni lista. Una funzione molto importante poichè è
37 necessario resettare le liste ogni volta che il parser valida una frase. */
38 void clear() {
39 T = NULL;
40 Exp = NULL;
41 Phrase = NULL;
42 }
43
44 /* La funzione put_testo inserisce una stringa in una lista.
45 Gli argomenti sono:
46 - TT, un doppio puntatore alla lista per consentire l'aggiunta di nuovi elementi;
47 - str, la stringa da inserire nel campo tes del nuovo elemento che viene inserito
48 nella lista. */
49 void put_testo(testo **TT, char *str)
50 {
51 testo *punt, *t_el;
52 t_el = (testo *)malloc(sizeof(testo));
53 t_el->tes = (char *)strdup(str);
54 t_el->next = NULL;
55 if(*TT == NULL)
56 *TT = t_el;
57 else
58 {
59 punt = *TT;
60 while(punt->next != NULL)
61 punt = punt->next;
62 punt->next = t_el;
63 }
64 }
65
66 /* La funzione get_testo stampa a video tutti i valori di una lista.
67 Gli argomenti sono:
68 - TT, un puntatore alla lista;
69 - nome, una stringa d'utilità per etichettare la stampa, in modo tale da indentificare
70 la lista. */
71 void get_testo(testo *TT, char *nome)
72 {
73 testo *punt = TT;
74
75 printf("/* * * * * * %s * * * * * */\n\n", nome);
76 while(punt != NULL) {

```

```

73 printf("Elemento: %s\n", punt->tes);
74 punt = punt->next;
75 }
76 printf("\n/* */\n\n");
77 }
78
79 /* La funzione countelements conta il numero di elementi presenti in una lista.
80 L'argomento è:
81 - T, un puntatore alla lista;
82 Restituisce il numero degli elementi. */
83 int countelements(testo *T) {
84 testo *punt = T;
85 int value = 0;
86
87 while(punt != NULL) {
88 value++;
89 punt = punt->next;
90 }
91 return value;
92 }
93
94 /* La funzione isnumeric stabilisce se una stringa è costituita da soli numeri,
95 ossia se si tratta di un numero o no.
96 L'argomento è:
97 - str, la stringa da analizzare;
98 Restituisce 1 se la stringa è un numero, 0 altrimenti. */
99 int isnumeric(char *str)
100 {
101 //prelevo il primo carattere.
102 char *c = (char *)strndup(str, 1);
103 //se è uguale a "-", è in analisi un numero negativo.
104 if(strcmp(c, "-") == 0) {
105 //scarta il segno "-".
106 c = (char *)strdup(str + 1);
107 //per ogni carattere verifica che sia un numero.
108 while(*c)
109 {
110 //se anche un solo carattere non è un numero, restituisce zero e
interrompe il ciclo.
111 if(!isdigit(*c))
112 return 0;
113 c++;
114 }
115 //se il primo carattere non è "-", è in analisi un numero maggiore o uguale a zero.
116 } else {
117 //per ogni carattere verifica che sia un numero.
118 while(*str)
119 {
120 //se anche un solo carattere non è un numero, restituisce zero e
interrompe il ciclo.
121 if(!isdigit(*str))
122 return 0;
123 str++;
124 }
125 }
126 return 1;
127 }

```