POLITECNICO DI BARI

I FACOLTÀ DI INGEGNERIA
CORSO DI LAUREA IN INGEGNERIA INFORMATICA
Dipartimento di Elettrotecnica ed Elettronica

TEMA D'ANNO
IN
"COMPILATORI E INTERPRETI"

BASIC256-ANALYZER

Analizzatore lessicale, sintattico e semantico

Docente:

Chiar.mo Prof. Giacomo Piscitelli

Studente:

Michele D'AGOSTINO

ANNO ACCADEMICO 2010/2011

Indice

"La nostra conoscenza, se paragonata alla realtà, è primitiva e infantile. Eppure è il bene più grande di cui disponiamo"

Albert Einstein

Prefazione

Basic256-Analyzer è un software che consente di effettuare l'analisi lessicale, sintattica e semantica di un codice scritto nel linguaggio di programmazione Basic256, il quale rappresenta un dialetto del più noto Basic.

Il BASIC (***Beginner's All purpose Symbolic Instruction Code***) è un linguaggio di programmazione di alto livello sviluppato nel 1964, presso l'Università di Dartmouth, per il calcolatore GE-255 sotto la direzione di J.Kemeny e T. Kurtz. Fu progettato per essere un linguaggio da imparare agevolmente, enfatizzando la semplicità d'uso piuttosto che la potenza espressiva, e facilmente trasportabile verso calcolatori differenti dal GE-255. L'acronimo stesso vuole sottolineare il target verso il quale si rivolgeva questo tipo di linguaggio: nacque per poter essere utilizzato anche dai principianti (il 75% degli studenti dell'Università di Dartmouth era iscritto a facoltà umanistiche).

Col passare degli anni furono sviluppate svariate versioni: una delle più famose è l'Altair-Basic sviluppato da Bill Gates e Paul Allen inizialmente per l'Altair 8800; l'Altair BASIC, in seguito alla rescissione del contratto di esclusiva con MITS, fu commercializzato come Microsoft BASIC dalla Microsoft, una società fondata da Gates ed Allen nel 1975, anche per altre piattaforme hardware. Del Microsoft BASIC fu rilasciata anche una versione denominata MBASIC per il sistema operativo CP/M, uno dei più diffusi dell'epoca. Grazie alla popolarità di questo sistema, usato su macchine di successo quali l'Osborne 1, anche la popolarità del Microsoft BASIC crebbe e Microsoft iniziò a rilasciare sempre più versioni del suo interprete.

Nel corso degli anni sono comparsi diversi dialetti Basic, ognuno dei quali sviluppa dei requisiti specifici per l'ambiente su cui dovrà girare il programma.

Il caso di studio verte su uno di questi dialetti, il BASIC256 (o KidBASIC), un progetto open-source nato per avvicinare i bambini alla programmazione. Esso utilizza le strutture di controllo tradizionali, come gosub, for/next, if, goto, che consentono ai bambini di conoscere come funziona un qualunque flusso di istruzioni. Ha inoltre un interprete che consente di visualizzare i risultati ottenuti e le figure descritte attraverso l'utilizzo di funzioni appositamente implementate.

CAPITOLO 1
INTRODUZIONE

1.1 Sintassi del Basic256

Un programma scritto in Basic256 consiste in una successione di statement, separati da newline, eseguiti nel loro ordine. Gli elementi principali della sintassi sono i seguenti:

- ✓ Costante numerica: una qualunque successione numerica, eventualmente preceduta dal segno meno per indicare i numeri negativi.
- ✓ Stringa: una successione di zero o più caratteri delimitati da " ".
- ✓ Variabile: le variabili che contengono valori numerici devono iniziare con una lettera, seguita da un qualsiasi numero di caratteri alfanumerici, e posso essere utilizzate in maniera intercambiabile con le costanti numeriche; le variabili che contengono stringhe seguono le stesse regole di composizione delle variabili numeriche ma devono terminare col simbolo $. Anch'esse possono essere usate in maniera intercambiale con le stringhe di caratteri.
- ✓ Array: sono allocati tramite il comando DIM e possono contenere numeri o stringhe. L'accesso agli elementi del vettore è effetuato per mezzo delle parentesi quadre, ricordando che il primo elemento del vettore è in posizione 0 (es: pippo[3] indica il 4° elemento del vettore di nome pippo).
- ✓ Operatori: consentono soprattutto le operazioni aritmetiche, le operazioni di assegnazione e concatenazione delle stringhe.

Order of Operations		
Level	**Operators**	**Category/Description**
1	( )	Grouping
2	^	Exponent
3	- ~	Unary Minus and Bitwise Negation (NOT)
4	* / \	Multiplication and Division
5	%	Integer Remainder (Mod)
6	+ -	Addition, Concatenation, and Subtraction
7	& |	Bitwise And and Bitwise Or

8	< ⇐ > >= = <>	Comparison (Numeric and String)
9	NOT	Unary Not
10	AND	Logical And
11	OR	Logical Or
12	XOR	Logical Exclusive Or

- ✓ Comandi: servono per l'esecuzione del programma stesso e consentono il controllo del flusso dati. Saranno trattati, con l'ausilio della BNF, nell'appendice.

Date le vaste specifiche grammaticali e sintattiche del Basic256, sono stati ridotti il lessico e la sintassi del linguaggio prendendo in considerazione i principali costrutti sintattici e ignorando i comandi aventi una sintassi simile a quella delle istruzioni analizzate.

1.2 Tools utilizzati

Per la realizzazione del Basic256-Analyzer mi sono avvalso dei seguenti tools:

- ✓ *Flex 2.5.4a*: scanner generator per la creazione dell'analizzatore lessicale.
- ✓ *Bison 2.4.1*: parser generator per la realizzazione dell'analizzatore sintattico
- ✓ *MinGW* (***Min***imalist ***G***NU for ***W***indows): ambiente di sviluppo per il linguaggio C, distriubuito per applicazioni Windows Microsoft.

1.3 Progetto di un compilatore

I programmi responsabili della traduzione da un linguaggio sorgente al linguaggio macchina sono detti *traduttori*. I traduttori si distinguono in:

1) Compilatori
2) Interpreti
3) Traduttori ibridi

Un *compilatore* è un programma che traduce un programma scritto in un linguaggio sorgente in uno logicamente equivalente scritto in un linguaggio target. Il linguaggio sorgente è di solito un linguaggio di alto livello (nel nostro caso il Basic256), mentre il linguaggio target è il linguaggio macchina (generalmente il linguaggio macchina dell'architettua di calcolo "ospitante"). In pratica, il compilatore traduce le operazioni scritte rispettando la grammatica del linguaggio sorgente in operazioni computazionali appartenenti alla macchina ospitante.

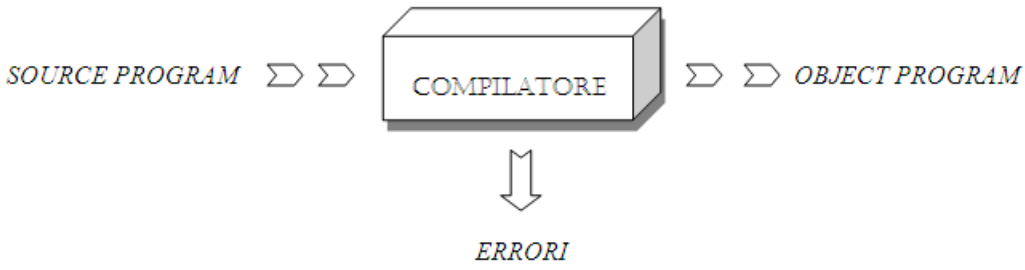


Figura 1: schema di un compilatore.

I compilatori attuali dividono l'operazione di compilazione in due stadi principali: il *front-end* (fase di analisi) e il *back-end* (fase di sintesi); nello stadio di front-end il compilatore traduce il sorgente in un linguaggio intermedio (di solito interno al compilatore), mentre nello stadio di back-end avviene la generazione del codice oggetto. Tali compilatori sono detti "*compilatori a 2 passi*".

1.3.1 Front-end & Back-end

Il front-end di un compilatore è costituito dai seguenti moduli:

- L'*analizzatore lessicale* (o *scanner*) raggruppa i caratteri letti dal file sorgente in unità lessicali, dette token. L'analizzatore lessicale riceve quindi in input uno stream di caratteri e restituisce in output un stream di token. Per la definizione dei token riconosciuti dallo scanner si utilizzano solitamente le espressioni regolari. Lo scanner è solitamente implementato mediante una macchina a stati finiti.
  Lex e Flex sono tools utilizzati per la generazione automatica di scanners.

- L'*analizzatore sintattico* (o *parser*), invece, raggruppa i token in frasi grammaticali e ne verifica la correttezza sintattica tramite la costruzione di un albero di derivazione (o *parse*

tree). Per definire le regole sintattiche del programma sorgente riconosciuto dal parser si utilizzano solitamente grammatiche libere dal contesto (Context-free grammars).
Yacc e Bison sono tools per la generazione automatica di parsers.

- L'*analizzatore sematico* verifica la presenza di errori semantici nel programma sorgente e acquisisce l'informazione sui tipi (*Type checking*) che verrà usata nella fase successiva di generazione del codice. L'output generato da questa fase è un *parse tree annotato*; per descrivere la semantica statica del programma sorgente vengono usate grammatiche ad attributi. Questa fase è combinata con il parser: durante il parsing, le informazioni (sulle variabili) utili per il context-checking sono immagazzinate nella *Symbol Table*.
- Il *generatore di codice intermedio*, infine, trasforma il parse tree in codice oggetto.

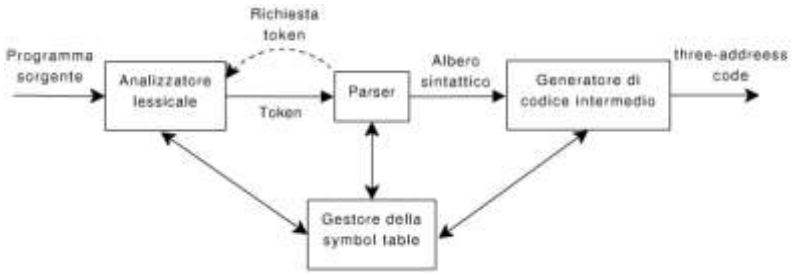


Figura 2: tipico schema di organizzazione del front-end di un compilatore (tratto da Aho, Lam, Sethi e Ullman, 2007).

Lo stadio di back-end, invece, è indipendente dal linguaggio sorgente ma dipende dall'architettura di calcolo; esso si occupa di ottimizzare il programma che si sta compilando (sotto forma di linguaggio intermedio) e, successivamente, di tradurlo in codice macchina per una specifica architettura. La descrizione dello stadio di back-end risulta essere alquanto esigua poiché lo scopo di questo progetto è la realizzazione dello stadio di front-end del compilatore per Basic256.

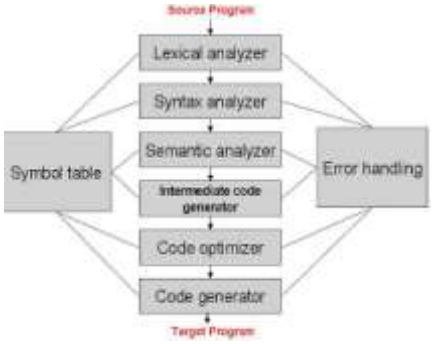


Figura 3: struttura completa di un compilatore.

CAPITOLO 2
ANALISI LESSICALE

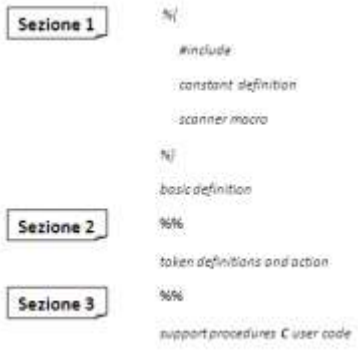
2.1 Flex input

Esistono 3 approcci per realizzare uno scanner: *realizzazione procedurale*, *realizzazione tabulare interpretata* e *realizzazione automatica mediante scanner generator*. In questo contesto, mi sono avvalso di uno scanner generator cioè un programma in grado di generare automaticamente un analizzatore lessicale. Il tool FLEX (**F**ast **LEX**ical analyzer generator), della Free Software Foundation, originariamente scritto in C da Vern Paxson nel 1987, è un free software, frequentemente usato con il parser generator Bison.

Flex utilizza per la specifica dell'analizzatore lessicale le espressioni regolari, che sono un formalismo più efficiente delle grammatiche ma meno potente. La distinzione tra grammatiche e espressioni regolari sta nel fatto che le espressioni regolari non sono in grado di riconoscere strutture sintattiche ricorsive, mentre ciò non è un problema per le grammatiche. Una struttura sintattica come le parentesi bilanciate, che richiede che le parentesi aperte siano nello stesso numero di quelle chiuse, non può essere riconosciuta da un analizzatore lessicale e per questo scopo si ricorre all'analizzatore sintattico. Invece costanti numeriche, identificatori o keyword vengono normalmente riconosciute dall'analizzatore lessicale.

Flex prende come input un file di testo (generalmente denominato *scanner.l*) contenente definizioni, di regole e codice C di supporto per l'analisi lessicale, fornendo in uscita un file sorgente C denominato generalmente *lex.yy.c* il quale implementa un automa a stati finiti deterministico (DFA) che consente di riconoscere i token.

L'input di Lex viene diviso in tre parti distinte, separate da %%:



Parti di codice possono essere inserite sia nella prima parte (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.
La **prima sezione** 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. Di seguito viene riportato un esempio:

and [Aa][Nn][Dd]

In questa sezione, se lo scanner fosse usato in combinazione col parser, andrebbe inserita la direttiva *#include parser.tab.h*, che consente di "includere" il file generato dal parser generator e contenente la definizione dei token multi-carattere ai fini dell'analisi sintattica.
La **seconda sezione** contiene le regole, sotto forma di coppie

pattern(espressione regolare) *azione*,

che vengono riconosciute dall'analizzatore lessicale. Alcune utilizzano le definizioni di base della prima sezione, racchiudendone il nome tra parentesi graffe. Associata ad ogni pattern c'è un'azione, espressa mediante qualsiasi statement in codice C, che viene eseguita ogni volta che l'espressione regolare corrispondente viene riconosciuta:

{and} *{ return AND; }*

I pattern devono essere non indentati e le azioni devono iniziare sulla stessa riga in cui termina l'espressione regolare; pattern e azioni devono essere separati tramite spazi o tabulazioni. Qualora il codice comprenda più di una istruzione o occupi più di una linea, dovrà essere racchiuso tra parentesi graffe. Quando, invece, l'azione è nulla (espressa con il carattere ';'), il token incontrato viene semplicemente scartato.
Quindi, ogni volta che l'analizzatore lessicale trova una sequenza di simboli riconosciuta dall'espressione regolare, indicata in una regola, esercita l'azione corrispondente.

Questo comportamento però potrebbe dar luogo a due tipologie di ambiguità lessicali:

1. la parte iniziale di una sequenza di caratteri riconosciuta da un' espressione regolare è riconosciuta anche da una seconda espressione regolare;
2. la stessa sequenza di caratteri è riconosciuta da due espressioni regolari distinte.

Per risolvere le ambiguità lessicali sono state stabilite due convenzioni:

1. quando la parte iniziale di una sequenza di caratteri è riconosciuta da due espressioni regolari viene eseguita l'azione associata all'espressione regolare che ha riconosciuto la sequenza più lunga;
2. quando la stessa sequenza di simboli è riconosciuta da due espressioni regolari distinte è eseguita l'azione corrispondente all'espressione regolare dichiarata per prima nel file sorgente di Flex.

Quindi, date le regole di risoluzione delle ambiguità lessicali, è buona norma definire prima le regole per le parole chiave e poi quelle per gli identificatori.
Di seguito è riportato un esempio di conflitto:

%%
for {return FOR_CMD;}
format {return FORMAT_CMD;}
[a-z]+ {return GENERIC_ID;}

Supponendo che la stringa di ingresso sia "*format*", lo scanner ritornerà il valore FORMAT_CD, preferendo la seconda regola alla prima perché descrive una sequenza più lunga e la seconda regola alla terza perché definita prima nel file sorgente.
La **terza sezione**,infine, 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 omesso. Tutte le righe presenti in questa sezione del programma sorgente sono ricopiate nel file *lex.yy.c* generato da Flex.
Ad esempio, nel file *scanner.y* relativo al progetto da me sviluppato viene definita una funzione che consente di rilevare e gestire gli errori lessicali.

2.2 *Flex output*

Se Flex non è combinato con Bison, compilando il file *scanner.l* si ottiene come risultato il file lex.yy.c, un programma C, privo di main() e contenente la routine di scanning ***yylex()*** assieme ad altre routine ausiliari e macro. Se, invece, si intende combinare l'analisi lessicale con quella sintattica, il file lex.yy.c conterrà un main in cui è descritta la funzione yylex().

```
2204  #if YY_MAIN
2205  int main()
2206      {
2207      yylex();
2208      return 0;
2209      }
2210  #endif
```

Figura 4: porzione del file lex.yy.c

La funzione yylex() viene richiamata ripetutamente dalla funzione principale del parser, yyparse(), ogni volta che è richiesto un nuovo token: quando il programma generato da Flex è in esecuzione, la funzione yylex() ad ogni invocazione legge il suo input un carattere alla volta (da sinistra a destra) da un puntatore a file FILE * chiamato *yyin*, finché trova il più lungo prefisso di input (quello sarà il corrente *lessema*[1]) che realizza il matching con uno dei pattern della sezione delle regole di traduzione. Il testo corrispondente (che rappresenta un token) viene reso disponibile attraverso il puntatore a carattere *yytext* e la sua lunghezza viene memorizzata nell'intero *yyleng*[2]. Vengono quindi eseguite le azioni corrispondenti al pattern per il quale avviene il match; quando viene ritornato un token, la funzione yylex() termina ma sarà richiamata dal parser quando avrà bisogno di un altro token.

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.

Ogni chiamata a yylex() restituisce un valore intero che rappresenta il tipo della categoria del token.

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.

[1] Sequenza di caratteri nel programma sorgente che realizza il "matching" con il pattern di un certo token.

[2] 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, che contiene la sua lunghezza. Operando su tali variabili si possono definire azioni più complesse e passare al parser, attraverso la funzione *new_string*, il testo letto dall'input.

Al termine del file di input, yylex() invoca la funzione yywrap() che può essere usata per continuare a leggere da un ulteriore file. Se yywrap() ritorna 0, si assume che yyin stia puntando ad un nuovo input file e la scansione continua attraverso la yylex(); se, invece, la funzione yywrap() restituisce il valore 1, lo scanner termina poiché il valore 1 corrisponde all'assenza di ulteriori file in input da analizzare

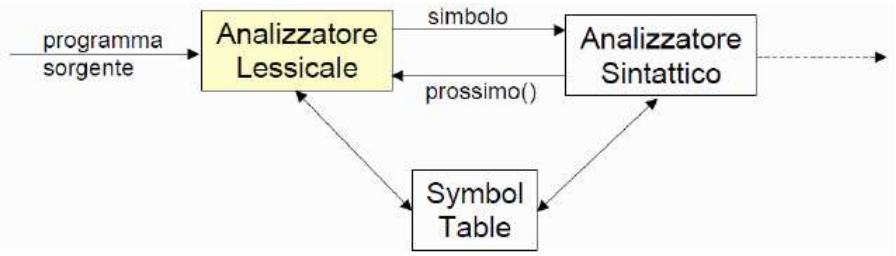


Figura 5: interazione tra Flex e Bison.

2.3 File scanner.l

```
%{

#include "parser.tab.h"
#include <string.h>     /*per utilizzare printf() e strdup()*/

void erroreLessicale(int);

int linenumber;

%}

/* definizione dei nomi dei token */

and [Aa][Nn][Dd]
black [Bb][Ll][Aa][Cc][Kk]
blue  [Bb][Ll][Uu][Ee]
cls   [Cc][Ll][Ss]
clg   [Cc][Ll][Gg]
close [Cc][Ll][Oo][Ss][Ee]
color [Cc][Oo][Ll][Oo][Rr]
colour [Cc][Oo][Ll][Oo][Uu][Rr]
clear [Cc][Ll][Ee][Aa][Rr]
cyan  [Cc][Yy][Aa][Nn]
dark  [Dd][Aa][Rr][Kk]
dim   [Dd][Ii][Mm]
end   [Ee][Nn][Dd]
```

for [Ff][Oo][Rr]

gosub [Gg][Oo][Ss][Uu][Bb]

goto [Gg][Oo][Tt][Oo]

gray [Gg][Rr][AaEe][Yy]

green [Gg][Rr][Ee][Ee][Nn]

if [Ii][Ff]

input [Ii][Nn][Pp][Uu][Tt]

instr [Ii][Nn][Ss][Tt][Rr]

key [Kk][Ee][Yy]

length [Ll][Ee][Nn][Gg][Tt][Hh]

mid [Mm][Ii][Dd]

next [Nn][Ee][Xx][Tt]

not [Nn][Oo][Tt]

open [Oo][Pp][Ee][Nn]

or [Oo][Rr]

orange [Oo][Rr][Aa][Nn][Gg][Ee]

pause [Pp][Aa][Uu][Ss][Ee]

print [Pp][Rr][Ii][Nn][Tt]

purple [Pp][Uu][Rr][Pp][Ll][Ee]

read [Rr][Ee][Aa][Dd]

red [Rr][Ee][Dd]

rem [Rr][Ee][Mm]

reset [Rr][Ee][Ss][Ee][Tt]

return [Rr][Ee][Tt][Uu][Rr][Nn]

sound [Ss][Oo][Uu][Nn][Dd]

step [Ss][Tt][Ee][Pp]

then [Tt][Hh][Ee][Nn]

to [Tt][Oo]

toint [Ii][Nn][Tt]

tostring [Ss][Tt][Rr][Ii][Nn][Gg]

white [Ww][Hh][Ii][Tt][Ee]

write [Ww][Rr][Ii][Tt][Ee]

xor [Xx][Oo][Rr]

yellow [Yy][Ee][Ll][Ll][Oo][Ww]

abs [Aa][Bb][Ss]

ceil [Cc][Ee][Ii][Ll]

cos [Cc][Oo][Ss]

floor [Ff][Ll][Oo][Oo][Rr]

pi [Pp][Ii]

rand [Rr][Aa][Nn][Dd]

sin [Ss][Ii][Nn]

tan [Tt][Aa][Nn]

whitespace [\t]+

nws [^ \t\n]+

integer [0-9]+

floatnum [0-9]\.?[0-9]**

string \"[^\"\n]\"*

*variable [a-zA-Z][a-zA-Z0-9]**

stringvar [a-zA-Z][a-zA-Z0-9]\$*

*varnum ([0-9])+[a-zA-Z][A-Za-z0-9_]**

stringnum ([0-9])+[a-zA-Z][A-Za-z0-9_]\$*

label {variable}:

blankline ^{whitespace}[\n]*

```
%%


{toint}        { return TOINT; }
{tostring}     { return TOSTRING; }
{length}       { return LENGTH; }
{mid}          { return MID; }
{instr}        { return INSTR; }
{and}          { return AND; }
{or}           { return OR; }
{xor}          { return XOR; }
{not}          { return NOT; }
{print}            { return PRINT; }
{dim}              { return DIM; }
{cls}              { return CLS; }
{clg}              { return CLG; }
{sound}        { return SOUND; }
{color}            { return SETCOLOR; }
{colour}           { return SETCOLOR; }


{clear}            { yylval.number = 0; return COLOR; }
{white}            { yylval.number = 1; return COLOR; }
{black}            { yylval.number = 2; return COLOR; }
{red}              { yylval.number = 3; return COLOR; }
{dark}{red}        { yylval.number = 4; return COLOR; }
{green}            { yylval.number = 5; return COLOR; }
{dark}{green}      { yylval.number = 6; return COLOR; }
{blue}             { yylval.number = 7; return COLOR; }
```

```
{dark}{blue}     { yylval.number = 8; return COLOR; }
{cyan}           { yylval.number = 9; return COLOR; }
{dark}{cyan}     { yylval.number = 10; return COLOR; }
{purple}         { yylval.number = 11; return COLOR; }
{dark}{purple}   { yylval.number = 12; return COLOR; }
{yellow}         { yylval.number = 13; return COLOR; }
{dark}{yellow}   { yylval.number = 14; return COLOR; }
{orange}         { yylval.number = 15; return COLOR; }
{dark}{orange}   { yylval.number = 16; return COLOR; }
{gray}           { yylval.number = 17; return COLOR; }
{dark}{gray}     { yylval.number = 18; return COLOR; }
{goto}           { return GOTO; }
{if}             { return IF; }
{then}           { return THEN; }
{for}            { return FOR; }
{to}             { return TO; }
{step}           { return STEP; }

{ceil}         { return CEIL; }
{floor}        { return FLOOR; }
{abs}          { return ABS; }
{sin}          { return SIN; }
{cos}          { return COS; }
{tan}          { return TAN; }
{rand}         { return RAND; }
{pi}           { return PI; }

{next}           { return NEXT; }
{open}           { return OPEN; }
```

```
{read}          { return READ; }
{write}         { return WRITE; }
{close}         { return CLOSE; }
{reset}         { return RESET; }
{input}       { return INPUT; }
{key}        { return KEY; }
{gosub}       { return GOSUB; }
{return}      { return RETURN; }
{pause}       { return PAUSE; }

{rem}        {return REM;}

{end}        { return END; }
">="         { return GTE; }
"<="         { return LTE; }
"<>"         { return NE; }
"+"          { return PLUS; }
"-"          { return '-'; }
"*"          { return '*'; }
"/"          { return '/'; }
"^"          { return '^'; }
"="          { return '='; }
"<"          { return '<'; }
">"          { return '>'; }
","          { return ','; }
";"          { return ';'; }
":"          { return ':'; }
"("          { return '('; }
")"          { return ')'; }
```

```
"{"             { return '{'; }
"}"             { return '}'; }
"["             { return '['; }
"]"             { return ']'; }
"$"             { return '$'; }

"\n"            { linenumber++;
                  return '\n'; }

{label} {

        yylval.string = yytext;

        return LABEL;
      }

{integer}       { yylval.number = atoi(yytext); return INTEGER; }

{floatnum}      { yylval.floatnum = atof(yytext); return FLOAT; }

{string}        {
                                        int len = strlen(yytext);
                                        yylval.string = strdup(yytext + 1);
                  yylval.string[len - 2] = 0;
                  return STRING; }

{variable}      {

                  yylval.string = yytext;
```

```
            return VARIABLE;
          }

{stringvar}    {

              yylval.string = yytext;

            return STRINGVAR;
          }

{varnum}    {

            yylval.string = yytext;

            return VARNUM;
          }

{stringnum}    {

            yylval.string = yytext;

            return STRINGNUM;
          }


{whitespace}    /* ignora lo spazio a fine riga */
```

```
.                       { printf("char: %s\n", yytext);
                                                erroreLessicale(2);
                                                }


L?\"(\\.|[^\\"\n])*     {erroreLessicale(1);}   /* mancanza di chiusura delle virgolette nelle stringhe di caratteri */

%%


int
yywrap(void) {
  return 1;
}

void erroreLessicale(int idErrore)
   {
        char messaggioErrore[1024];

                        if(idErrore == 1)
        {
           sprintf(messaggioErrore, " Errore LESSICALE: virgolette aperte ma non chiuse\n");

        }

                        if(idErrore == 2)
        {
           sprintf(messaggioErrore, " Errore LESSICALE: carattere non riconosciuto\n");
```

```
    }

                yyerror1(messaggioErrore);

}
```

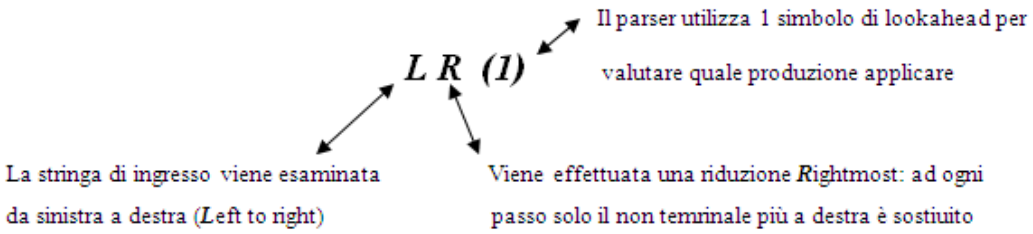
CAPITOLO 3
ANALISI SINTATTICA

3.1 Bison input

Un parser deve riconoscere la struttura sintattica di una stringa di ingresso e stabilire se la sequenza è una frase ammessa dalla sintassi; la sintassi è fornita in termini di regole di produzione di una Context Free Grammar (CFG), di una BNF, o di diagrammi sintattici. Dopo aver riconosciuto le stringhe di token valide (cioè generabili tramite la grammatica), l'analizzatore sintattico crea l'albero sintattico (*parse tree*) in cui ogni foglia è un terminale e ogni nodo rappresenta un simbolo non temrinale. Esistono 3 tipi di parser per grammatiche context-free:

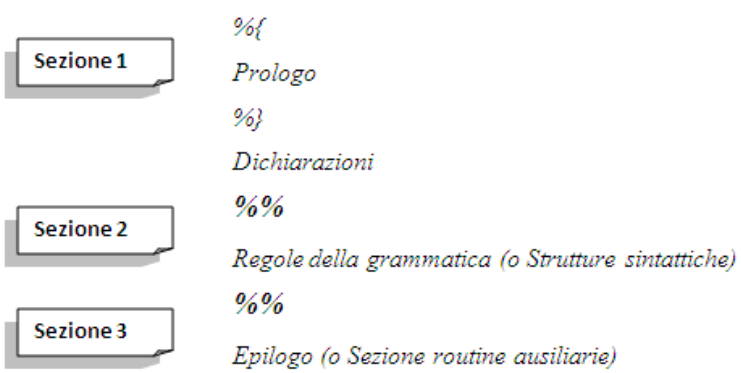
1) *parser universali*: riescono ad analizzare qualsiasi grammatica ma non sono efficienti;
2) *parser top-down*: l'albero sintattico della stringa viene generato a partire dalla radice (assioma) scendendo fino alle foglie (simboli terminali della stringa di input);
3) *parser bottom-up*: l'albero sintattico della stringa di input viene generato a partire dalle foglie, risalendo fino alla radice.

Così come per lo scanner, esistono strumenti automatici per costruire automaticamente anche il parser: esempi di parser generator sono Yacc e Bison; questi tool sono molto utili perché forniscono informazioni diagnostiche: ad esempio, in caso di grammatica ambigua, riescono a determinare dove si crea l'ambiguità.
Il tool da me utilizzato per generare il parser è *Bison*: consente di descrivere le produzioni della grammatica del linguaggio da riconoscere e le azioni da intraprendere per ogni produzione.
Bison è un parser di tipo bottom-up e, pertanto, opera su una grammatica LR(1):



Il file di input a Bison è un file di testo, generalmente nominato *parser.y*, composto da 3 sezioni distinte e separate da **%%**.

.



Nel "*Prologo*" sono contenuti gli *#inlcude* delle librerie necessarie, nonché la definizione di alcune variabili che serviranno per realizzare al meglio le azioni sintattiche: questo codice C sarà copiato all'interno del file *parser.tab.c* che implementa il parser.

La sezione delle *regole* è l'unica obbligatoria. I caratteri di spaziatura (blank, tab e newline) vengono ignorati, mentre i commenti sono racchiusi, come in C, tra i simboli /* e */.

Inoltre, nel *prologo*

Le *dichiarazioni* servono a definire i simboli della grammatica e i valori semantici ad essi associati:

- *%token* <*simbolo*> : dichiara un simbolo terminale
- *%left* <*simbolo*> : dichiara l'assocciatività a sinistra di un simbolo terminale
- *%right* <*simbolo*> : dichiara l'assocciatività a destra di un simbolo terminale
- *%nonassoc* <*simbolo*> : dichiara la non associatività di un simbolo terminale
- *%type* <*simbolo*> : dichiara un simbolo non terminale
- *%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 contenente le *regole* della grammatica 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 più simboli sia terminali che non terminali.

I simboli **:** e **;** sono separatori. Nel caso la grammatica presenti più produzioni per lo stesso simbolo non terminale A, queste possono essere scritte senza ripetere il non terminale usando 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 {}.

```
163
164 boolexpr: stringexpr '=' stringexpr  { printf("Riconosciuto un EGUAL tra stringhe\n", linenumber); }
165         | stringexpr NE stringexpr   { printf("Riconosciuto un NOT EGUAL tra stringhe\n", linenumber); }
166         | floatexpr '=' floatexpr    { printf("Riconosciuto un EGUAL tra numeri\n", linenumber); }
167         | floatexpr NE floatexpr     { printf("Riconosciuto un NOT EGUAL tra numeri\n", linenumber); }
168         | floatexpr '<' floatexpr    { printf("Riconosciuto un LESS tra numeri\n", linenumber); }
```

Figura 6: esempio di regole grammaticali dal file parser.y

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

Le azioni possono scambiare dei valori con il parser tramite delle pseudo-variabili introdotte dai simboli: $$, $1, $2, … La pseudo-variabile $$ si riferisce al valore dell'attributo associato al non terminale che costituisce la testa della produzione mentre le pseudovariabili $$_i$$ si riferiscono al valore associato all'*i*-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.

Figura 7: esempio di definizione di regole grammaticali.

La terza e ultima sezione 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 yylval. L'analizzatore lessicale può essere implementato in due possibili modi:
  I. si scrive esplicitamente all'interno della sezione Funzioni ausiliarie una procedura yylex() in C;
  II. si usa un programma separato (nel mio caso Flex) per generare l'analizzatore lessicale e in questo caso 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 I/O di un input. All'interno del file parser.y si può 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.

3.2 Bison output

Compilando il file *parser.y* si ottiene il file *parser.tab.c* in cui viene definita la funzione yyparse(); tale funzione è l'implentazione del parser: yyparse() riceve i token, esegue le azioni e termina quando incontra la fine dell'input o un errore di sintassi irreversibile. 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, in particolare, 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 symboli 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 corrispettivo symbol number.

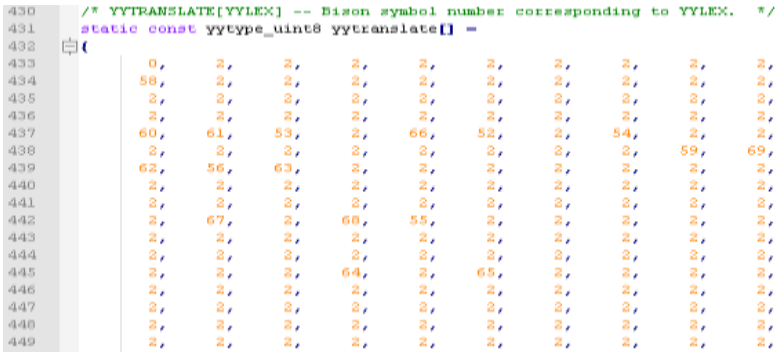


Figura 8: porzione della tabella YYTRANSLATE contenuta nel file parser.tab.c

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

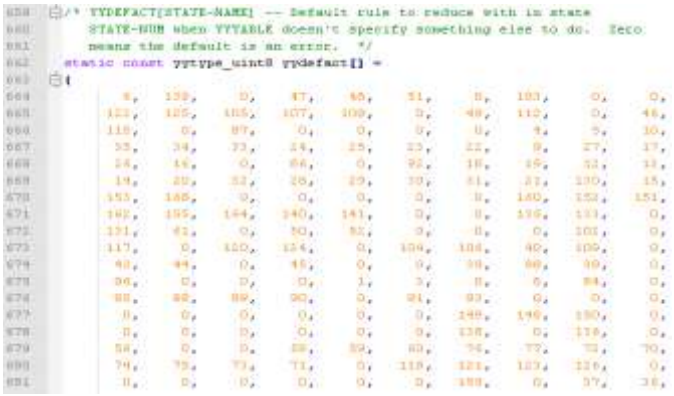


Figura 9: porzione della tabella YYDEFACT contenuta nel file parser.tab.c

3.3 File parser.y

```
%{

   #include <stdlib.h>
   #include <stdio.h>
   #include <string.h>
   #include "ST.h"

   int count = 0;
   int yylex(void);
   int yyerror(const char *);
         int yyerror1(const char *);
   extern int linenumber;

%}

%expect 1

/* definizione dei token associati ai simboli terminali */

%token PRINT INPUT KEY
%token CLS CLG
%token IF THEN FOR TO STEP NEXT
%token OPEN READ WRITE CLOSE RESET
%token GOTO GOSUB RETURN REM END SETCOLOR
%token GTE LTE NE PLUS
%token DIM
%token TOINT TOSTRING LENGTH MID INSTR
```

```
%token AND OR XOR NOT
%token PAUSE SOUND
%token CEIL FLOOR RAND SIN COS TAN ABS PI

%union
{
  int number;
  double floatnum;
  char *string;   /* Identifiers */
}


%token <number> LINENUM
%token <number> INTEGER
%token <floatnum> FLOAT

%token <string> STRING
%token <string> VARIABLE
%token <string> STRINGVAR

%token <string> VARNUM
%token <string> STRINGNUM

%token <number> COLOR
%token <string> LABEL

/*definizione dei non terminali */

%type <floatnum> floatexpr
```

```
%type <string> stringexpr

/*definizione delle precedenze tra gli operatori*/

%left '-'
%left PLUS
%left '*'
%left '/'
%left '^'
%left AND
%left OR
%left XOR
%right '='
%nonassoc UMINUS


%%

program: validline '\n' program
       | validline '\n'
;

validline: ifstmt      {}
        | compoundstmt {}
        | /*empty*/    { printf("Riconosciuta riga vuota\n"); }
        ;

ifstmt: ifexpr THEN compoundstmt { printf("Riconosciuta istruzione di diramazione\n"); }
         | ifexpr THEN  {yyerror("Errore SINTATTICO: manca corpo del THEN"); }
```

```
        | ifexpr {yyerror(" Errore SINTATTICO sullo statement if: THEN mancante\n"); }
;



compoundstmt: statement
            | compoundstmt':' statement
;

statement: gotostmt
        | gosubstmt
        | returnstmt
        | printstmt
        | numassign
        | stringassign
        | forstmt
        | nextstmt
        | colorstmt
        | inputstmt
        | endstmt
        | clearstmt
        | dimstmt
        | pausestmt
        | arrayassign
        | strarrayassign
        | openstmt
        | writestmt
        | closestmt
        | resetstmt
        | soundstmt
```

```
                |remstmt
                |labelstmt
                |errless
;

errless: STRINGNUM '=' stringexpr    {yyerror("Errore LESSICALE: i nomi degli identificatori devono iniziare con una lettera\n");  }
        |VARNUM '=' floatexpr       {yyerror("Errore LESSICALE: i nomi degli identificatori devono iniziare con una lettera\n");  }
;

labelstmt: VARIABLE':'              { printf("Riconosciuta LABEL\n", linenumber); }
         | VARIABLE error           {yyerror("Errore SINTATTICO: label/variabile non completa\n");  }
;

remstmt: REM STRING                 { printf("Riconosciuta linea di commenti\n"); }
;

dimstmt: DIM VARIABLE '(' floatexpr ')'    { printf("Allocazione di un array contenente numeri\n");}
       |DIM STRINGVAR '(' floatexpr ')'   { printf("Allocazione di un array contenente stringhe \n");}

       |DIM VARIABLE                {yyerror("Errore SINTATTICO: specificare la dimensione del vettore\n");  }
              |DIM STRINGVAR                {yyerror("Errore SINTATTICO: specificare la dimensione del vettore\n");  }
;

pausestmt: PAUSE floatexpr              { printf("Riconosciuta istruzione di pausa\n"); }
         | PAUSE                    {yyerror("Errore SINTATTICO:specificare la durata della pausa\n");  }
;
```

```
clearstmt: CLS    { printf("Riconosciuta istruzione di pulizia dello schermo testuale\n"); }
        | CLG    { printf("Riconosciuta istruzione di pulizia dello schermo grafico\n"); }
;

endstmt: END      { printf("Riconosciuta istruzione di fine\n"); }
;

ifexpr: IF compoundboolexpr    { printf("Riconosciuto uno statement IF con condizione booleana\n"); }
      | IF                {yyerror1("Errore SINTATTICO:condizione dell'IF mancante\n"); }
;

compoundboolexpr: boolexpr                          { printf("Riconosciuta un'espressione booleana\n"); }
            |compoundboolexpr AND compoundboolexpr   { printf("Riconosciuto un AND logico\n"); }
            |compoundboolexpr OR compoundboolexpr    { printf("Riconosciuto un OR logico\n"); }
            |compoundboolexpr XOR compoundboolexpr   { printf("Riconosciuto un XOR logico\n"); }
            |NOT compoundboolexpr %prec UMINUS       { printf("Riconosciuto un NOT logico\n"); }
            | '(' compoundboolexpr ')'
            | compoundboolexpr AND  {yyerror1(" Errore SINTATTICO: manca l'operatore a destra
dell'AND\n");  }
            | compoundboolexpr OR   {yyerror1(" Errore SINTATTICO: manca l'operatore a destra
dell'OR\n");  }
            | compoundboolexpr XOR  {yyerror1(" Errore SINTATTICO: manca l'operatore a destra
dell'XOR\n");  }
            | NOT               {yyerror1(" Errore SINTATTICO: manca l'operando dopo il NOT\n");  }
;

boolexpr: stringexpr '=' stringexpr  { printf("Riconosciuto un EGUAL tra stringhe\n"); }
      | stringexpr NE stringexpr   { printf("Riconosciuto un NOT EGUAL tra stringhe\n"); }
      | floatexpr '=' floatexpr    { printf("Riconosciuto un EGUAL tra numeri\n"); }
```

```
    | floatexpr NE floatexpr     { printf("Riconosciuto un NOT EGUAL tra numeri\n"); }
    | floatexpr '<' floatexpr    { printf("Riconosciuto un LESS tra numeri\n"); }
    | floatexpr '>' floatexpr    { printf("Riconosciuto un GREAT tra numeri\n"); }
    | floatexpr GTE floatexpr    { printf("Riconosciuto un GREAT AND EGUAL tra numeri\n"); }
    | floatexpr LTE floatexpr    { printf("Riconosciuto un LESS AND EGUAL tra numeri\n"); }
                | floatexpr '='          {yyerror1(" Errore SINTATTICO: manca l'operatore a destra
dell'uguale\n");  }
    | stringexpr '='          {yyerror1(" Errore SINTATTICO: manca l'operatore a destra dell'uguale\n");  }
    | floatexpr NE            {yyerror1(" Errore SINTATTICO: manca l'operatore a destra di <>\n");  }
    | stringexpr NE           {yyerror1(" Errore SINTATTICO: manca l'operatore a destra di <>\n");  }
    | floatexpr '<'           {yyerror1(" Errore SINTATTICO: manca l'operatore a destra di <\n");  }
    | floatexpr '>'           {yyerror1(" Errore SINTATTICO: manca l'operatore a destra di >\n");  }
    | floatexpr GTE           {yyerror1(" Errore SINTATTICO: manca l'operatore a destra del >=\n");  }
    | floatexpr LTE           {yyerror1(" Errore SINTATTICO: manca l'operatore a destra del <= \n");  }
;

immediatestrlist: '{' stringlist '}'    {printf("Riconosciuta lista di stringhe inserite nell'array \n"); }
;

stringlist: stringexpr
      | stringexpr ',' stringlist
;

immediatelist: '{' floatlist '}'       {printf("Riconosciuta lista di valori inseriti nell'array \n"); }
;

floatlist: floatexpr
      | floatexpr ',' floatlist
;
```

```
strarrayassign:  strfloatstmt stringexpr          { printf("Riconosciuta l'assegnazione,nel vettore, di una stringa\n"); }

        | strass_stmt immediatestrlist      { printf("Riconosciuta l'assegnazione di un array di stringhe\n"); }

        | strass_stmt                  {yyerror("Errore SINTATTICO nell'assegnazione della stringa\n");  }

                | STRINGVAR                    {yyerror("Errore SINTATTICO nell'assegnazione della stringa\n");  }

        | strass_stmt  floatexpr          {yyerror("Errore SINTATTICO: assegnazione di un valore numerico ad una stringa\n");  }

                |strfloatstmt floatexpr                                {yyerror("Errore SINTATTICO: il vettore non puo' contenere valori numerici\n");  }

                                ;


stringassign: strass_stmt stringexpr       { printf("Riconosciuta l'assegnazione della stringa \n"); }

;


arrayassign: varfloatstmt floatexpr      { printf("Riconosciuta l'assegnazione,nel vettore, di un numero\n"); }

        | varass_stmt immediatelist   { printf("Riconosciuta l'assegnazione di un array di numeri\n"); }

        | varass_stmt              {yyerror("Errore SINTATTICO nell'assegnazione della variabile\n");  }

                | varfloatstmt stringexpr         {yyerror("Errore SINTATTICO: il vettore non puo' contenere stringhe\n");}

;


numassign: varass_stmt floatexpr { printf("Riconosciuta assegnazione numerica\n");}

;


strfloatstmt: STRINGVAR '[' floatexpr ']' '='  {install1($1,V_CHARS);}

        |STRINGVAR '[' VARIABLE ']' '='  {}

;
```

```
strass_stmt: STRINGVAR '='  {install($1,V_CHARS);}

;


varfloatstmt: VARIABLE '[' floatexpr ']' '='  {install($1,V_CHARS);}

;


varass_stmt: VARIABLE '='  {install($1,V_CHARS);}

;


forstmt: FOR VARIABLE '=' floatexpr TO floatexpr                { printf("Riconosciuto ciclo FOR\n"); }

      | FOR VARIABLE '=' floatexpr TO floatexpr STEP floatexpr  { printf("Riconosciuto ciclo FOR\n"); }

      | FOR VARIABLE '=' floatexpr                              {yyerror("Errore SINTATTICO: manca condizione
finale del ciclo FOR\n");}

;



nextstmt: NEXT VARIABLE    { printf("Riconosciuta NEXT\n"); }

      | NEXT          {yyerror("Errore SINTATTICO nello statement NEXT\n");  }

;


gotostmt: GOTO VARIABLE     { printf("Riconosciuto salto incondizionato\n"); }

      | GOTO          {yyerror("Errore SINTATTICO nello statement GOTO\n");  }

;


gosubstmt: GOSUB VARIABLE   { printf("Riconosciuto salto incondizionato\n");}

      | GOSUB         {yyerror("Errore SINTATTICO nello statement GOSUB\n");  }

;


returnstmt: RETURN         { printf("Riconosciuto RETURN\n"); }
```

```
;

colorstmt: SETCOLOR COLOR                    { printf("Riconosciuta impostazione testuale
colore\n"); }

        | SETCOLOR '(' COLOR ')'              { printf("Riconosciuta impostazione testuale colore\n");
}

                | SETCOLOR '-' INTEGER                  { printf("Riconosciuta impostazione colore
trasparente\n"); }

                | SETCOLOR '(' '-' INTEGER ')'          { printf("Riconosciuta impostazione colore
trasparente\n"); }

                | SETCOLOR '(' INTEGER ',' INTEGER ',' INTEGER ')'    { printf("Riconosciuta
impostazione RGB del colore\n"); }

                | SETCOLOR  INTEGER ',' INTEGER ',' INTEGER        { printf("Riconosciuta
impostazione RGB del colore\n"); }

        | SETCOLOR                      {yyerror("Errore SINTATTICO: specificare il colore");  }

                | SETCOLOR VARIABLE             {yyerror("Errore SINTATTICO: colore non
riconosciuto\n");}

                | SETCOLOR '(' INTEGER ',' INTEGER ')'          {yyerror("Errore SINTATTICO: manca
una componente RGB\n");}

                | SETCOLOR  INTEGER ',' INTEGER          {yyerror("Errore SINTATTICO:
manca una componente RGB\n");}

                ;




soundstmt: SOUND '(' floatexpr ',' floatexpr ')'  { printf("Riconosciuto SOUND\n"); }

        | SOUND floatexpr ',' floatexpr        { printf("Riconosciuto SOUND\n"); }

                | SOUND                 {yyerror("Errore SINTATTICO: non definite frequenza e
durata\n");  }

;
```

```
openstmt: OPEN '(' stringexpr ')' { printf("Riconosciuta operazione di apertura di un file\n"); }
      | OPEN stringexpr        { printf("Riconosciuta operazione di apertura di un file\n"); }
                  | OPEN '(' stringexpr     {yyerror("Errore SINTATTICO: manca parentesi tonda\n");  }
;

writestmt: WRITE '(' stringexpr ')'  { printf("Riconosciuta operazione di scrittura in coda al file\n"); }
      | WRITE stringexpr         { printf("Riconosciuta operazione di scrittura in coda al file\n"); }
                  | WRITE '(' stringexpr      {yyerror("Errore SINTATTICO: manca parentesi tonda\n");  }

;

closestmt: CLOSE         { printf("Operazione di chiusura del file\n"); }
      | CLOSE '(' ')'  { printf("Operazione di chiusura del file\n"); }
                  | CLOSE '('      {yyerror("Errore SINTATTICO: manca parentesi tonda\n");  }
;

resetstmt: RESET         { printf("Riconosciuto RESET del file aperto\n"); }
      | RESET '(' ')'  { printf("Riconosciuto RESET del file aperto\n"); }
;

inputstmt: inputexpr ',' STRINGVAR                { printf("Riconosciuta la lettura di un messaggio\n"); }
      | inputexpr ',' STRINGVAR '[' floatexpr ']'  { printf("Riconosciuta la lettura di un messaggio\n"); }
                  | inputexpr ',' VARIABLE                { printf("Riconosciuta la lettura di un messaggio\n");
}
                   | inputexpr                        {yyerror("Errore SINTATTICO: variabile di destinazione
dell'input mancante\n");  }
                  ;

inputexpr: INPUT stringexpr                  { printf("Riconosciuto INPUT\n"); }
      | INPUT '(' stringexpr ')'             { printf("Riconosciuto INPUT \n"); }
```

```
                ;

floatstring: floatexpr PLUS stringexpr {}
                        |stringexpr PLUS floatexpr {}
                        | floatstring PLUS floatexpr {}
                        | floatstring PLUS stringexpr {}
;

printstmt: PRINT stringexpr             { printf("Riconosciuto PRINT\n"); }
                |PRINT floatstring              { printf("Riconosciuto PRINT\n"); }
        | PRINT '(' stringexpr ')'      { printf("Riconosciuto PRINT \n"); }
        | PRINT floatexpr                       { printf("Riconosciuto PRINT \n"); }
        | PRINT stringexpr ';'                  { printf("Riconosciuto PRINT, non andando a capo\n"); }
        | PRINT '(' stringexpr ')' ';'          { printf("Riconosciuto PRINT, non andando a capo \n"); }
        | PRINT floatexpr  ';'                  { printf("Riconosciuto PRINT, non andando a capo \n"); }
                | PRINT                         {yyerror("Errore SINTATTICO: manca l'espressione nello
statement PRINT\n");  }
;


floatexpr: '(' floatexpr ')'            {printf("Riconosciuto valore numerico\n");}
        | floatexpr PLUS floatexpr      { printf("Riconosciuta operazione di addizione\n"); }
        | floatexpr '-' floatexpr       { printf("Riconosciuta operazione di sottrazione\n"); }
        | floatexpr '*' floatexpr       { printf("Riconosciuta operazione di moltiplicazione\n"); }
        | floatexpr '/' floatexpr       { printf("Riconosciuta operazione di divisione\n"); }
        | floatexpr '^' floatexpr       { printf("Riconosciuta operazione di elevamento a potenza\n"); }
        | '-' FLOAT %prec UMINUS        { printf("Riconosciuto un float con un segno meno\n"); }
        | '-' INTEGER %prec UMINUS      { printf("Riconosciuto un intero con un segno meno\n");}
        | '-' VARIABLE %prec UMINUS     { printf("Riconosciuta una variabile numerica con un segno
meno\n");}
```

```
| FLOAT                          { printf("Riconosciuto un float\n");}

| INTEGER                        { printf("Riconosciuto un intero \n"); }

| KEY                            { printf("Riconosciuto KEY\n");}

| VARIABLE '[' floatexpr ']'     { printf("Riconosciuta variabile\n"); }

| VARIABLE                      { printf("Riconosciuta variabile\n"); }

| TOINT '(' floatexpr ')'       { printf("Riconosciuta conversione da espressione in intero \n"); }

| TOINT '(' stringexpr ')'      { printf("Riconosciuta conversione da stringa in intero \n"); }

| LENGTH '(' stringexpr ')'     { printf("Riconosciuto LENGHT\n"); }

| INSTR '(' stringexpr ',' stringexpr ')'  { printf("Riconosciuto INSTR\n"); }

        | CEIL '(' floatexpr ')'        { printf("Riconosciuta operazione CEIL\n"); }

| FLOOR '(' floatexpr ')'                       { printf("Riconosciuta operazione
FLOOR\n"); }

| SIN '(' floatexpr ')'                         { printf("Riconosciuta operazione SIN\n"); }

| COS '(' floatexpr ')'                         { printf("Riconosciuta operazione COS\n");
}

| TAN '(' floatexpr ')'                         { printf("Riconosciuta operazione TAN\n");
}

| ABS '(' floatexpr ')'                         { printf("Riconosciuta operazione ABS\n"); }

| RAND                                                  { printf("Riconosciuta
operazione RAND\n"); }

        | PI                    { printf("Riconosciuta la costante pigreco\n"); }

        | VARNUM                {yyerror1("Errore SINTATTICO: denominazione errata
della variabile\n");  }

        ;



stringexpr: stringexpr PLUS stringexpr          {printf("Riconosciuta concatenzaione tra
stringhe\n");}

| STRING                        {printf("Riconosciuta stringa\n");}

| STRINGVAR '[' floatexpr ']'    {}

| STRINGVAR                     {printf("Riconosciuta stringa\n");}
```

```
        | TOSTRING '(' floatexpr ')'                {printf("Riconosciuto TOSTRING\n");}
        | MID '(' stringexpr ',' floatexpr ',' floatexpr ')'  {printf("Riconosciuto MID\n");}
        | READ '(' ')'                            {printf("Lettura da file\n");}
        | READ                                    {printf("Lettura da file\n");}
                    | STRINGNUM                                {yyerror1("Errore SINTATTICO:
denominazione errata della variabile\n");  }
;



%%

/* la funzione yyerror viene chiamata da yyparse() per la gestione di eventuali errori sintattici */

        int yyerror(const char *msg) {

   printf("RIGA %d == %s", linenumber-1, msg);
        count++;
   return 0;
}

int yyerror1(const char *msg) {

   printf("RIGA %d == %s", linenumber, msg);
        count++;
   return 0;
}



main( int argc, char *argv[] )
```

```
{
  extern FILE *yyin;
  ++argv; --argc;
  int i;
  for(i=0;i<argc;i++){

    printf("ANALISI DEL FILE %d: %s\n\n", i+1, argv[i]);
    yyin = fopen( argv[i], "r" );
    linenumber = 1;

    yyparse ();
    printf("\n");
  }

  if(count == 0){ printf("---Parsing BASIC-256 riuscito---\n");}

    else printf("---Parsing BASIC-256 fallito---\n");
}
```

CAPITOLO 4
ERROR RECOVERY

4.1 Errori lessicali e sintattici

La diagnostica degli errori lessicali è stata realizzata tenendo conto delle modalità di funzionamento dello scanner generator Flex e del parser generator Bison; in Basic256 due tipologie di errori lessicali sono molto comuni:

1. la mancanza di chiusura delle virgolette nelle stringhe di caratteri,
2. la denominazione scorretta degli identificatori, i quali possono iniziare unicamente con una lettera ma non con una cifra.

Per identificare e gestire gli errori di tipo 1 è stata inserita la seguente regola nel file *scanner.l*:

```
212   L?\"(\\.|[^\\"\n])*        {erroreLessicale(1);}
```

Figura 10: regola per gestire la mancanza di chiusura delle virgolette nelle stringhe di caratteri.

in cui *erroreLessicale(1)* è una funzione definita nella sezione delle procedure di supporto e che, in base al valore passato come parametro, mostra un messaggio con la descrizione dell'errore lessicale.

```
245 void erroreLessicale(int idErrore)
246     {
247             char messaggioErrore[1024];
248
249             if(idErrore == 1)
250             {
251                 sprintf(messaggioErrore, " Errore LESSICALE: virgolette aperte ma non chiuse\n");
252
253             }
```

Figura 11: funzione, definita in scanner.l, per la gestione degli errori lessicali.

Invece, gli errori di tipo 2 sono stati gestiti attraverso l'analisi sintattica: sono state definite le espressioni regolari relative alle variabili che cominciano con un numero e che quindi non sono ammissibili nel linguaggio, poi sono stati definiti i corrispondenti token ed infine, tramite l'ausilio

della grammatica, sono state aggiunte le regole nel file *parser.y* per gestire questa tipologia di errori lessicali.

```
72  varnum        ([0-9])+[a-zA-Z][A-Za-z0-9_]*
```

```
191 {varnum}      {
192
193                   yylval.string = yytext;
194
195                   return VARNUM;
196               }
197
```

```
73  stringnum     ([0-9])+[a-zA-Z][a-zA-Z0-9]*\$
```

```
198 {stringnum}     {
199
200                   yylval.string = yytext;
201
202                   return STRINGNUM;
203               }
```

```
53  %token <string> VARNUM
54  %token <string> STRINGNUM
```

```
124 errless: STRINGNUM '=' stringexpr   {yyerror("Errore LESSICALE: i nomi deg
125        |VARNUM '=' floatexpr        {yyerror("Errore LESSICALE: i nomi deg
```

Figura 12: gestione degli errori lessicali di tipo 2 (denominazione scorretta degli identificatori).

Di seguito è riportato un breve programma Basic256 strutturato in modo tale da evidenziare la rilevazione e la gestione degli errori lessicali;

```
1  rem "esempio di errori lessicali"
2
3  1pippo$ = "cane"
4
5  11topolino = 3*5
6
7  stringa$ = "esempio di errore lessicale
8
9
```

Figura 13: programma per la gestione degli errori lessicali.

L'output generato dalle precedenti righe di codice è:

```
Riconosciuta linea di commenti
Riconosciuta riga vuota
Riconosciuta stringa
RIGA 3 == Errore LESSICALE: i nomi degli identificatori devono iniziare con una
lettera
Riconosciuta riga vuota
Riconosciuto un intero
Riconosciuto un intero
Riconosciuta operazione di moltiplicazione
RIGA 5 == Errore LESSICALE: i nomi degli identificatori devono iniziare con una
lettera
Riconosciuta riga vuota
RIGA 7 ==  Errore LESSICALE: virgolette aperte ma non chiuse

RIGA 7 == Errore SINTATTICO nell'assegnazione della stringa
Riconosciuta riga vuota

---Parsing BASIC-256 fallito---

C:\Documents and Settings\Michele\Desktop\COMPILATORI\Mio_Progetto\DEFINITIVO>pa
use
Premere un tasto per continuare . . .
```

Figura 14: output relativo al programma in figura 13.

Si noti come l'errore lessicale generato alla riga 7, derivante dalla mancata chiusura delle virgolette, generi in cascata un errore sintattico poiché il parser non riconosce l'istruzione corretta di assegnazione una stringa.

Nella

```
C:\WINDOWS\system32\cmd.exe

C:\Documents and Settings\Michele\Desktop\COMPILATORI\Mio_Progetto\DEFINITIVO>cd
 C:\Documents and Settings\Michele\Desktop\COMPILATORI\Mio_Progetto\DEFINITIVO

C:\Documents and Settings\Michele\Desktop\COMPILATORI\Mio_Progetto\DEFINITIVO>Fl
ex scanner.l

C:\Documents and Settings\Michele\Desktop\COMPILATORI\Mio_Progetto\DEFINITIVO>Bi
son -d parser.y

C:\Documents and Settings\Michele\Desktop\COMPILATORI\Mio_Progetto\DEFINITIVO>gc
c -c parser.tab.c

C:\Documents and Settings\Michele\Desktop\COMPILATORI\Mio_Progetto\DEFINITIVO>gc
c -c lex.yy.c

C:\Documents and Settings\Michele\Desktop\COMPILATORI\Mio_Progetto\DEFINITIVO>gc
c parser.tab.o lex.yy.o -o michele.exe

C:\Documents and Settings\Michele\Desktop\COMPILATORI\Mio_Progetto\DEFINITIVO>mi
chele.exe prova.txt
ANALISI DEL FILE 1: prova.txt
```

realizzazione del progetto si è utilizzato un approccio di tipo "*error production*" per il rilevamento e la gestione degli errori sintattici: consente di non interrompere la fase di parsing quando vengono rilevati degli errori sintattici grazie all'utilizzo di produzioni che estendono la grammatica, in modo tale da generare errori comuni.

Ad esempio nel codice si è inserita la produzione che prevede l'esecuzione della funzione *yyerror()* passando come argomento il messaggio di errore lessicale:

```
220  nextstmt: NEXT VARIABLE    { printf("Riconosciuta NEXT\n"); }
221          | NEXT              {yyerror("Errore SINTATTICO nello statement NEXT");  }
222  ;
```

Figura 15: approccio di tipo "error production" per la gestione degli errori sintattici.

Di seguito si riporta un esempio di codice sintatticamente corretto (Fig.15) ed un esempio di codice sintatticamente scorretto (Fig.16), affiancati dai corrispondenti output:

```
1  rem "esempio di programma corretto"
2
3  input "inserire età",age
4
5  if (age >= 18) then print "maggiorenne"
6      if (age < 18) then print "minorenne"
7
8
```

```
C:\WINDOWS\system32\cmd.exe

C:\Documents and Settings\Michele\Desktop\COMPILATORI\Mio_Progetto\DEFINITIVO>ed
 C:\Documents and Settings\Michele\Desktop\COMPILATORI\Mio_Progetto\DEFINITIVO

C:\Documents and Settings\Michele\Desktop\COMPILATORI\Mio_Progetto\DEFINITIVO>Fl
ex scanner.l

C:\Documents and Settings\Michele\Desktop\COMPILATORI\Mio_Progetto\DEFINITIVO>Bi
son -d parser.y

C:\Documents and Settings\Michele\Desktop\COMPILATORI\Mio_Progetto\DEFINITIVO>gc
c -c parser.tab.c

C:\Documents and Settings\Michele\Desktop\COMPILATORI\Mio_Progetto\DEFINITIVO>gc
c -c lex.yy.c

C:\Documents and Settings\Michele\Desktop\COMPILATORI\Mio_Progetto\DEFINITIVO>gc
c parser.tab.o lex.yy.o -o michele.exe

C:\Documents and Settings\Michele\Desktop\COMPILATORI\Mio_Progetto\DEFINITIVO>mi
chele.exe prova.txt
ANALISI DEL FILE 1: prova.txt

Riconosciuta linea di commenti
Riconosciuta riga vuota
Riconosciuta stringa
Riconosciuto INPUT
Riconosciuta la lettura di un messaggio
Riconosciuta riga vuota
Riconosciuta variabile
Riconosciuto un intero
Riconosciuto un GREAT AND EQUAL tra numeri
Riconosciuta un'espressione booleana
Riconosciuto uno statement IF con condizione booleana
Riconosciuta stringa
Riconosciuto PRINT
Riconosciuta istruzione di diramazione
Riconosciuta variabile
Riconosciuto un intero
Riconosciuto un LESS tra numeri
Riconosciuta un'espressione booleana
Riconosciuto uno statement IF con condizione booleana
Riconosciuta stringa
Riconosciuto PRINT
Riconosciuta istruzione di diramazione
Riconosciuta riga vuota

---Parsing BASIC-256 riuscito---

C:\Documents and Settings\Michele\Desktop\COMPILATORI\Mio_Progetto\DEFINITIVO>pa
use
Premere un tasto per continuare . . .
```

Figura 16: codice sintatticamente corretto.

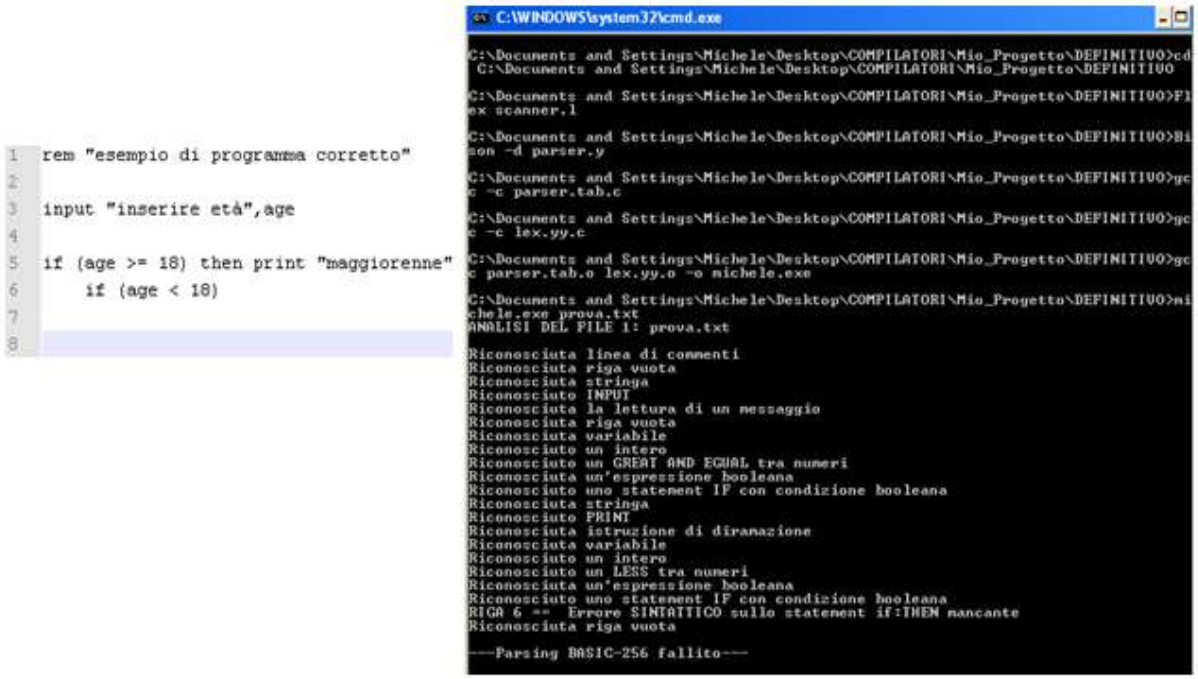


Figura 17: codice non sintatticamente corretto.

4.2 Errori semantici

L'analizzatore progettato, grazie a Bison, consente un controllo di tipo semantico ma solo per alcuni aspetti del linguaggio: l'analisi è stata rivolta essenzialmente alla definizione delle variabili. Il meccanismo previsto consente di riconoscere se il nome della variabile che si sta cercando di definire non sia già stato utilizzato all'interno dello stesso programma. Nel caso in cui si cerchi di ridefinire una variabile già assegnata in precedenza, il parser rileverà un errore semantico specificando che la variabile non può essere ridefinita. In caso contrario (la variabile viene assegnata per la prima volta) si provvederà all'instanziazione e memorizzazione nella *symbol table* delle informazioni relative alla funzione.

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

L'elemento basilare della lista ha la seguente struttura:

```
/*-------------------------------------------------------------------------
RECORD DELLA TABELLA DEI SIMBOLI
-------------------------------------------------------------------------*/
//un record della tabella dei simboli è una struttura avente i seguenti campi:
struct symrec
{
    char *name; /* nome del simbolo */
    st
    ti
};
```

Figura 18: struttura di un elemento della symble table.

prova.txt | parser.y | scanner.l | ST.h | compilatore.bat

```
1  rem "primo esempio di errore semantico"
2
3  dim z(5)
4
5  z=(1,3,5,7,9)
6
7  print z[0]+z[4]
8
9  z=(0,2,4,6,8)
10
11 rem "secondo esempio di errore semantico"
12
13 pippo$ = "esame"
14
15 print pippo$
16
17 pippo$ = "di compilatori"
18
```

Normalmente la tabella può prevedere un gran numero di parametri aggiuntivi, volti a particolareggiare i dati, in modo da approfondire l'analisi semantica. Ai fini del progetto si è comunque pensato di trascurare altri parametri, concentrandosi sugli attributi che servono a descrivere in chiave semantica le variabili.

Il primo campo della struttura è associato al nome della variabile, mentre il secondo campo è utilizzato per realizzare la lista concatenata di elementi (esso rappresenta infatti il puntatore all'elemento successivo della lista).

Per la gestione della symble table sono state previste le funzioni *putsym()* e *getsym()* che servono rispettivamente ad inserire dati nella lista (nella symbol table) e ad estrarre dati dalla lista.

Tali funzioni sono utilizzate nella funzione *install()*, chiamata dal parser quando viene matchata una produzione (cioè quando si ha una reduce), ed in corrispondenza di essa è prevista un'azione semantica:

Qui di seguito è mostrato un semplice esempio contenente gli errori semantici sopra descritti:

```
C:\WINDOWS\system32\cmd.exe
c -c lex.yy.c

C:\Documents and Settings\Michele\Desktop\COMPILATORI\Mio_Progetto\DEFI
c parser.tab.o lex.yy.o -o michele.exe

C:\Documents and Settings\Michele\Desktop\COMPILATORI\Mio_Progetto\DEFI
chele.exe prova.txt
ANALISI DEL FILE 1: prova.txt
```

Figura 19: programma con errori sematici.

Per quanto riguarda il "*type checking*", in fase di analisi semantica non è stato effettuato il controllo sulla dichiarazione degli idientificatori: il Basic256 consente di istanziare una variabile (di tipo numerico o di tipo stringa) senza la necessità che essa sia stata dichiarata in precedenza. Invece, è stato effettuato il controllo sulla coerenza dei tipi durante l'assegnazione delle variabili:

1. una variabile di tipo stringa non può contenere una variabile numerica
2. una variabile di tipo numerico non può contenere una successione di caratteri.

Le due precedenti regole, e quindi il controllo di coerenza dei tipi, si applicano anche nel caso in cui le variabili coinvolte nell'assegnazione siano degli array.

4.3 File ST.h

```
/* definiamo la variabile tipo_var, la quale può assumere uno dei qualsiasi valori indicati tra graffe*/
enum tipo_var {INT=1,CHAR,vet,V_INT,V_FLOAT,V_CHARS,ESPRESSIONE};
typedef enum tipo_var tipo_variabile;
/*con typedef creiamo un nuovo tipo di dato, ovvero tipo_variabile, il quale è di tipo tipo_var,
che a sua volta può assumere uno dei valori racchiusi tra graffe sopra*/

char
*tipo_var_char[]={"","INT","FLOAT","CHAR","vet","V_INT","V_FLOAT","V_CHARS","ESPRESSIONE"}
;
/*------------------------------------------------------------------------
RECORD DELLA TABELLA DEI SIMBOLI
------------------------------------------------------------------------*/
//un record della tabella dei simboli è una struttura avente i seguenti campi:
struct symrec
{
        char *name; /* nome del simbolo */
        struct symrec *next; /* puntatore a un campo della struttura */
        tipo_variabile type;
};
typedef struct symrec symrec;

symrec *sym_table = (symrec *)0; /* sym_table è un puntatore alla struttura symrec
                                                        (la tabella dei simboli) e lo inizializziamo a
zero. */

//le operazioni possibili sono: putsym, getsym


/*con la funzione putsym restituiamo il puntatore alla tabella dei simboli, che punta al nuovo record inserito.
il nuovo simbolo viene inserito in testa, dopo che gli è stata allocata la giusta memoria per contenerlo.*/
symrec * putsym (char *sym_name, int val)
{
        symrec *ptr; //dichiato ptr come puntatore ad un array di elementi symrec
        ptr = (symrec *) malloc (sizeof(symrec));
        ptr->name = (char *) malloc (strlen(sym_name)+1);
```

```
        strcpy (ptr->name,sym_name);
        ptr->type=val;
        ptr->next = (struct symrec *)sym_table;
        sym_table = ptr;
        return ptr;
}
```

/*con la funzione getsym restituiamo un puntatore alla tabella dei simboli. Come argomento, viene passato il simbolo che vogliamo prelevare dalla tabella dei simboli. Finché il puntatore non è nullo (cioè uguale a zero) viene scansionata tutta la tabella, finché il simbolo cercato non è uguale al simbolo che risiede nel record i-esimo della tabella dei simboli. Una volta trovato il simbolo, viene restituito il suo puntatore*/

```
symrec * getsym (char *sym_name)
{
        symrec *ptr;

           for ( ptr = sym_table; ptr!=(symrec *)0; ptr=(symrec *)ptr->next)
                    if (strcmp (ptr->name,sym_name) == 0)
                            return ptr;
           return 0;
}
```

/* con la funzione install effettuiamo un controllo semantico. L'obiettivo della funzione è evitare di ridefinire una variabile già definita. Passiamo come parametro il puntatore al simbolo appena immesso, il suo tipo e la sua dimensione

Creiamo un puntatore alla tabella dei simboli e mettiamo in tale puntatore, l'eventuale puntatore corrispondente al nuovo simbolo immesso. Se tale puntatore non viene trovato, significa che il simbolo non è presente nella tabella dei simboli. In questo caso, tramite la funzione putsym, inseriamo questo simbolo nella tabella. Se invece viene trovato, significa che abbiamo già inserito il nome di quel simbolo nella tabella*/

```
void install ( char *sym_name, int tipo_var)
{
        symrec *s;
        s = getsym (sym_name);
```

```
        if (s == 0) //cioè il simbolo non è stato trovato nella symble table
                s = putsym (sym_name,tipo_var);
        else {
                yyerror1( "Errore SEMANTICO:\n");
                printf("la variabile == %s= e' stata gia' definita\n",sym_name);
        }
}
```

CAPITOLO 5
FUNZIONAMENTO DEL BASIC256-ANALYZER

5.1 Comandi per la compilazione e l'avvio

Il progetto è stato sviluppato in modo tale che il Basic256-Analyzer possa funzionare in modalità testuale: dopo aver opportunamente installato i software Flex, Bison e Gcc, è possibile avviare dal prompt dei comandi il front-end del compilatore.
Tuttavia, prima di procedere alla compilazione e al successivo avvio dell'analizzatore, è necessario effettuare un'operazione preliminare che agevola il lavoro di compilazione da parte dell'utente.
L'operazione preliminare si effettua (per s.o. Windows XP) tramite la seguente procedura:

1. dal "*Pannello di controllo*" (in modalità 'visualizzazione classica'), fare doppio click sull'icona "*Sistema*";
2. a questo punto occorre cliccare su "*Avanzate*" e, nella finestra che si apre, accedere nella sezione "*Variabili d'ambiente*";
3. in questa nuova sezione sono mostrate le **variabili d'utente** e le **variabili di sistema**; ai fini del progetto, selezionare il tasto "*Nuovo*" nella sezione "*Variabili di sistema*": definiamo in tal modo una nuova variabile di sistema con i seguente requisiti

   - Nome: *Path*
   - Valore: *C:\Programmi\GnuWin32\bin;C:\MinGW\bin;C:\OpenCV2.2\bin;C:\MinGW\msys\1.0\bin*

4. infine cliccare su *ok* per rendere effettiva l'operazione.

In questa maniera sarà possibile compilare i file *scanner.l* e *parser.y* semplicemente selezionando la directory contenente questi ultimi, evitando di selezionare i path che contengono i software Flex, Bison e GCC fondamentali per la compilazione.
A questo punto, per compilare ed avviare l'analizzatore è necessario aprire il prompt dei comandi, posizionarsi nella cartella contenente i sorgenti scanner.l e parser.y tramite il comando cd (**c**hange **d**irectory) e digitare i seguenti comandi:

Comandi per la compilazione

Comandi da prompt	Input file	Output file
Flex scanner.l	*scanner.l*	*lex.yy.c*
Bison -d parser.y	*parser.y*	*parser.tab.c* *parser.tab.h*
gcc -c parser.tab.c	*parser.tab.c*	*parser.tab.o*
gcc -c lex.yy.c	*lex.yy.c*	*lex.yy.o*
gcc parser.tab.o lex.yy.o -o basic256.exe	*parser.tab.o* *lex.yy.o*	*basic256.exe*
basic256.exe prova.txt	*prova.txt*	*compilazione del file prova.txt*

Descrizione dei comandi

Comandi da prompt	Descrizione
Flex scanner.l	*Genera il file lex.yy.c*
Bison -d parser.y	*Genera il file parser.tab.c e anche il suo header parser.tab.o*
gcc -c parser.tab.c	*Compila e assembla il file parser.tab.c senza linkarlo, creando un file oggetto per ogni file sorgente specificato; in questo caso genera solo parser.tab.o*
gcc -c lex.yy.c	*Compila e assembla il file lex.yy.c senza linkarlo, creando un file oggetto per ogni file sorgente specificato; in questo caso genera solo lex.yy.o*
gcc parser.tab.o lex.yy.o -o basic256.exe	*Compila i file .o contenuti nella cartella e genera l'eseguibile "basic256.exe"*

CAPITOLO 6

TEST SU CODICI BASIC256

In questo capitolo verrano presentati una serie di programmi (scritti in Basic256) "ad hoc" cioè realizzati con l'intento di evidenziare il corretto funzionamento del front-end del compilatore, nonché di mostrare come avviene la gestione degli errori lessicali, sintattici e semantici.
Ciascun programma presentato nel seguito rappresenta in un certo senso un "*case test*" poiché consente di valutare la bontà del progetto da un punto di vista della rilevazione degli errori.
Un ambiente di sviluppo completo per Basic256 (dotato di editor integrato) analizza file con estensione ***.kbs***; in questo caso, invece, i programmi sono scritti all'interno di file di testo .txt e sono analizzati dal programma sorgente basic256.exe.

6.1 Esempi di programmi Basic256 corretti

In questo paragrafo sono presentati due brevi programmi, privi sia di errori lessicali che di errori sintattici, per evidenziare il normale funzionamento dell'analizzatore.

6.1.1 Codice n°1: statement GOSUB

In questo esempio si introduce il concetto di subroutine nell'ambito del costrutto "*gosub*": si tratta di un "pezzo" di codice che può essere chiamato da altre parti del programma per eseguire un task.
Di seguito è riportato il codice ed il rispettivo output generato dall'analizzatore:

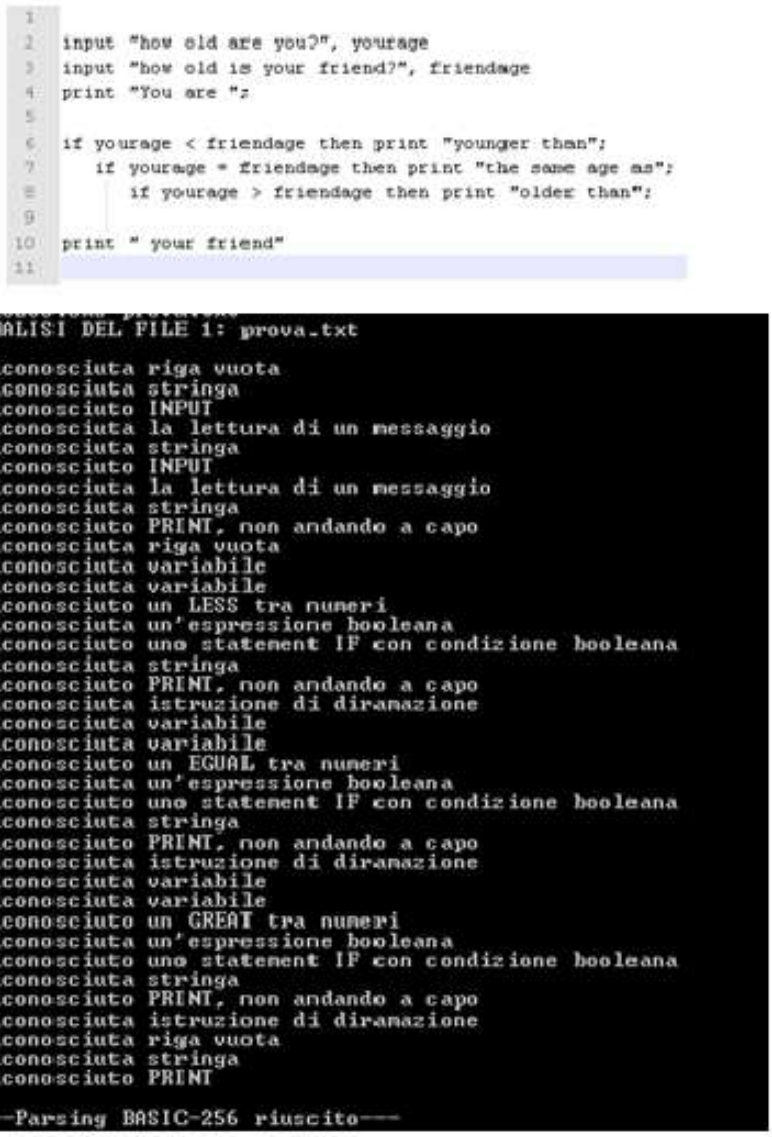
```
1  rem "gosubdemo.kbs"
2
3  gosub showline
4  print "hi"
5  gosub showline
6  print "there"
7  gosub showline
8  end
9
10 showline :
11 print "--------------------"
12 return
13
```

```
C:\Documents and Settings\Michele\Desktop\CO
sic256.exe prova.txt
ANALISI DEL FILE 1: prova.txt

Riconosciuta linea di commenti
Riconosciuto salto incondizionato
Riconosciuta stringa
Riconosciuto PRINT
Riconosciuto salto incondizionato
Riconosciuta stringa
Riconosciuto PRINT
Riconosciuto salto incondizionato
Riconosciuta istruzione di fine
Riconosciuta riga vuota
Riconosciuta LABEL
Riconosciuta stringa
Riconosciuto PRINT
Riconosciuto RETURN

---Parsing BASIC-256 riuscito---
```

Figura 20: primo esempio di codice scritto correttamente.

6.1.2 Codice n°2: statement IF-THEN

Il seguente programma mostra l'utilizzo del costrutto if-then per effettuare delle scelte: vengono utilizzati tre statemente di tipo if-then per valutare l'età del sottoscritto rispetto all'età di una seconda persona. Di seguito è riportato il codice ed il rispettivo output generato dall'analizzatore:



```
1
2  input "how old are you?", yourage
3  input "how old is your friend?", friendage
4  print "You are ";
5
6  if yourage < friendage then print "younger than";
7     if yourage = friendage then print "the same age as";
8        if yourage > friendage then print "older than";
9
10 print " your friend"
11
```

```
ANALISI DEL FILE 1: prova.txt

Riconosciuta riga vuota
Riconosciuta stringa
Riconosciuto INPUT
Riconosciuta la lettura di un messaggio
Riconosciuta stringa
Riconosciuto INPUT
Riconosciuta la lettura di un messaggio
Riconosciuta stringa
Riconosciuto PRINT, non andando a capo
Riconosciuta riga vuota
Riconosciuta variabile
Riconosciuta variabile
Riconosciuto un LESS tra numeri
Riconosciuta un'espressione booleana
Riconosciuto uno statement IF con condizione booleana
Riconosciuta stringa
Riconosciuto PRINT, non andando a capo
Riconosciuta istruzione di diramazione
Riconosciuta variabile
Riconosciuta variabile
Riconosciuto un EGUAL tra numeri
Riconosciuta un'espressione booleana
Riconosciuto uno statement IF con condizione booleana
Riconosciuta stringa
Riconosciuto PRINT, non andando a capo
Riconosciuta istruzione di diramazione
Riconosciuta variabile
Riconosciuta variabile
Riconosciuto un GREAT tra numeri
Riconosciuta un'espressione booleana
Riconosciuto uno statement IF con condizione booleana
Riconosciuta stringa
Riconosciuto PRINT, non andando a capo
Riconosciuta istruzione di diramazione
Riconosciuta riga vuota
Riconosciuta stringa
Riconosciuto PRINT

---Parsing BASIC-256 riuscito---
```

Figura 21: secondo esempio di codice scritto correttamente.

Si noti l'uso del comando PRINT seguito dal ";": consente di stampare a video una stringa di caratteri senza però andare a capo; l'intento del programma è quello di stampare sullo stdout un'unica stringa, sulla stessa linea, composta da 3 sottostringhe: una deriva dal print di riga 4, una dal print di riga 10 e una sottostringa deriva da uno dei print associato agli statement IF-THEN.

6.2 Esempi di programmi Basic256 con errori

In questo paragrafo vengono presentati una serie di programmi affetti da errori di tipo lessicale, di tipo sintattico e di tipo semantico.

6.2.1 Codice n°3: errori lessicali e sintattici

In questo esempio vengono utilizzati gli array per contenere delle stringhe di caratteri: per creare gli array si fa riferimento al comando *dim*. Di seguito è riportato il codice in Basic256 contenente sia errori sintattici sia errori lessicali:

```
1  rem "listoffriends.kbs"
2  print "make a list of my friends"
3  input "how many friends do you have?"
4
5  dim names$(n)
6
7  for i = 0
8  input "enter friend name ?", names$[i]
9  next i
10
11 cls
12 print "my friends
13 for i = 0 to n-1
14 print "friend number ";
15 print i + 1;
16 print " is " + names$[i]
17 next i
18
```

Figura 22: primo esempio di codice non scritto correttamente.

Più in dettaglio, tale codice contiene 3 errori sintattici ed 1 errore lessicale; uno dei tre errrori sintattici deriva, come vedremo, dalla presenza dell'errore lessicale. Analizziamo gli erorri presenti:

1) Riga 3 (*syntax error*) : non definita la variabile (di tipo stringa) di destinazione del messaggio.
2) Riga 7 (*syntax error*): manca la condizione finale del ciclo for cioè manca lo statement TO.
3) Riga 12 (*lexical error*): l'errore lessicale deriva dalla mancanza delle virgolette al termine della stringa; da ciò deriva un errore sintattico in quanto il parser non è in grado di riconoscere il costrutto corretto dello statement PRINT.

L'output, corrispondente al codice, dell'analizzatore sarà:

```
C:\Documents and Settings\Michele\Desktop\COMPILATORI\Mio_Progetto\DEFINITIVO>ba
sic256.exe prova.txt
ANALISI DEL FILE 1: prova.txt

Riconosciuta linea di commenti
Riconosciuta stringa
Riconosciuto PRINT
Riconosciuta stringa
Riconosciuto INPUT
RIGA 3 == Errore SINTATTICO: variabile di destinazione dell'input mancante
Riconosciuta riga vuota
Riconosciuta variabile
Allocazione di un array contenente stringhe
Riconosciuta riga vuota
Riconosciuto un intero
RIGA 7 == Errore SINTATTICO: manca condizione finale del ciclo FOR
Riconosciuta stringa
Riconosciuto INPUT
Riconosciuta variabile
Riconosciuta la lettura di un messaggio
Riconosciuta NEXT
Riconosciuta riga vuota
Riconosciuta istruzione di pulizia dello schermo testuale
RIGA 12 ==  Errore LESSICALE: virgolette aperte ma non chiuse

RIGA 12 == Errore SINTATTICO: manca l'espressione nello statement PRINT
Riconosciuto un intero
Riconosciuta variabile
Riconosciuto un intero
Riconosciuta operazione di sottrazione
Riconosciuto ciclo FOR
Riconosciuta stringa
Riconosciuto PRINT, non andando a capo
Riconosciuta variabile
Riconosciuto un intero
Riconosciuta operazione di addizione
Riconosciuto PRINT, non andando a capo
Riconosciuta stringa
Riconosciuta variabile
Riconosciuta concatenzaione tra stringhe
Riconosciuto PRINT
Riconosciuta NEXT

---Parsing BASIC-256 fallito---
```

Figura 23: output generato dal Basic256-analyzer, corrispondente al codice di figura 22.

6.2.2 Codice n°4: errori semantici

Nel programma seguente vengono evidenziati e gestiti due errori semantici dovuti al tentativo di riassegnare un valore ad una variabile " stringa" ed una lista di valori ad una variabile di tipo array:

```
1  rem "primo esempio di errore semantico"
2  a$ = string (10+13)
3  print a$
4  b$ = string(2 * pi)
5  print b$
6  b$ = "si sta commettendo un errore"
7  print b$
8
9  rem "secondo esempio di errore semantico"
10 dim vett(4)
11 vett={0,2,4,6}
12 for i=0 to 3
13 print vett[i]
14 next i
15
16 vett={1,3,5,7}
17
```

Figura 24: secondo esempio di codice non scritto correttamente.

L’analizzatore provvederà a segnalare l’errore semantico, indicando la variabile coinvolta:

Figura 25: output generato dal Basic256-analyzer, corrispondente al codice di figura 24.

6.2.3 Codice n°5: errori lessicali, sintattici e semantici

Per terminare la rassegna sulle varie tipologie di errori gestite dall'analizzatore, in questo paragrafo viene presentato un programma che consente il calcolo delle radici di un'equazione di secondo grado: al suo interno sono presenti tutte le tipologie di errori (lessicali, sintattici e semantici).

```
1
2   cls : rem "clear text output windows"
3   sol$= "soluzioni immaginarie"
4   2a=1
5   b=3
6   c=-4
7   delta = b*b-4*a*c
8
9   if (delta <0) then print 9sol$
10      if (delta>=0) then x1= (-b+ delta^(1/2))/2*a : x1= (-b - delta^(1/2))/2*a
11
12  print x1
13  print x2
14
```

Figura 26: secondo esempio di codice non scritto correttamente.

Gli errori presenti nel precedente codice sono evidenziati dall'analizzatore:

```
C:\Documents and Settings\Michele\Desktop\COMPILATORI\Mio_Progetto\DEFINITIVO>ba
sic256.exe prova.txt
ANALISI DEL FILE 1: prova.txt

Riconosciuta riga vuota
Riconosciuta istruzione di pulizia dello schermo testuale
Riconosciuta linea di commenti
Riconosciuta stringa
Riconosciuta l'assegnazione della stringa
Riconosciuto un intero
RIGA 4 == Errore LESSICALE: i nomi degli identificatori devono iniziare con una
lettera
Riconosciuto un intero
Riconosciuta assegnazione numerica
Riconosciuto un intero con un segno meno
Riconosciuta assegnazione numerica
Riconosciuta variabile
Riconosciuta variabile
Riconosciuta operazione di moltiplicazione
Riconosciuto un intero
Riconosciuta variabile
Riconosciuta operazione di moltiplicazione
Riconosciuta variabile
Riconosciuta operazione di moltiplicazione
Riconosciuta operazione di sottrazione
Riconosciuta assegnazione numerica
Riconosciuta riga vuota
Riconosciuta variabile
Riconosciuto un intero
Riconosciuto un LESS tra numeri
Riconosciuta un'espressione booleana
Riconosciuto uno statement IF con condizione booleana
RIGA 9 == Errore SINTATTICO: denominazione errata della variabile nello statemen
t PRINT
```

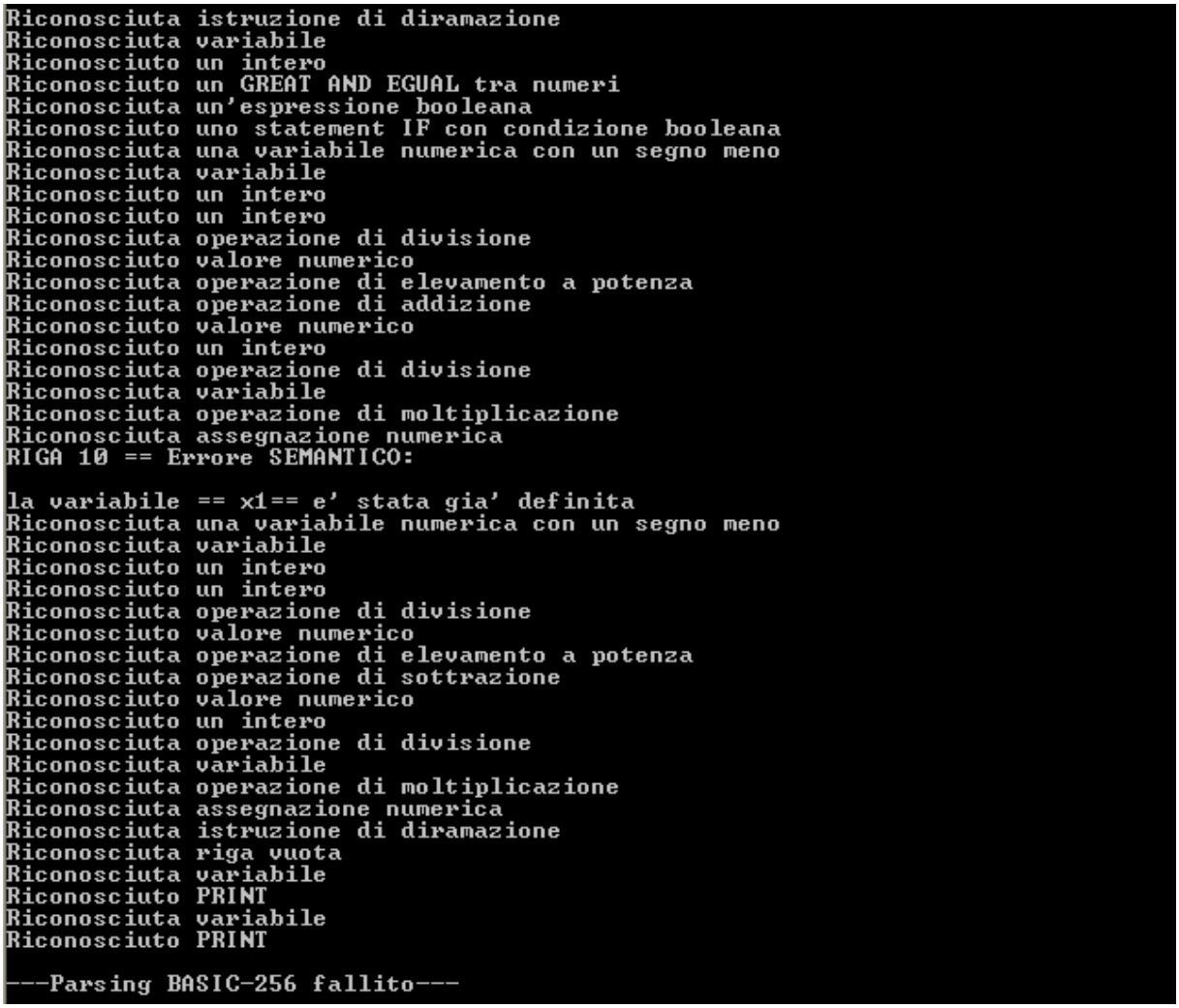


Figura 27: output generato dal Basic256-analyzer, corrispondente al codice di figura 26.

Conclusioni

In questa relazione è stato sviluppato un compilatore per un sottoinsieme di istruzioni del linuaggio Basic256, che analizza files eseguibili in ambiente DOS/Windows; il lavoro svolto mi ha permesso di apprendere alcune delle potenzialità di tools come Flex e Bison, free software alternativi, rispettivamente, a Lex e Yacc.

Tali software permettono la generazione automatica del codice C indispensabile per la realizzazione dell'analizzatore lessicale e sintattico. Il compilatore può essere utilizzato per scopi didattici e in ambiente accademico.

Il progetto del compilatore può essere completato e migliorato nel futuro aggiungendovi un controllo (nell'ambito degli errori semantici) sulla dimensione dei vettori e dei loro elementi, nonché nuove funzionalità quali, ad esempio, la creazione di una interfaccia grafica, o la produzione di files eseguibili per altri ambienti (calcolatori e/o sistemi operativi).

APPENDICE

BNF e commento alle produzioni

La BNF (**B**ackus–**N**aur **f**orm) è una notazione utilizzata per rappresentare in forma compatta la grammatica *context free* di un linguaggio; una regola BNF è composta da un simbolo nonterminale e da un'espressione (anzichè una semplice sequenza di terminali e nonterminali): la regola spiega, come una produzione, il modo di riscrivere il nonterminale, ma ha una sintassi molto più ricca. Per poterli distinguere dai non terminali, i simboli terminali sono scritti in grassetto maiuscolo. Di seguito è riportata la BNF di partenza utilizzata per la creazione dell'analizzatore sintattico:

program ::= validline \n | validline \n program

Un programma in Basic256 è composto da una o più validline.

validline ::= ifstmt | compoundstm | label

Una validline può essere composta da un'istruzione condizionale (if statement), da un'istruzione composta o da una label.

ifstmt ::= ifexpr THEN compoundstmt

Un'istruzione condizionale valuta la condizione booleana contenuta nell'ifexp: se tale espressione restituisce true, viene eseguita l'istruzione composta (compoundstmt), altrimenti viene ignorata e viene eseguita la linea successiva.

compoundstmt ::= statement | compoundstmt : statement

Un'istruzione composta è costituita da uno o più statement; nel caso in cui gli statement sono più d'uno, sono separati dai due punti (:).

statement ::= gotostmt | gosubstmt | returnstmt | printstmt| plotstmt| circlestmt| rectstmt| polystm | linestm | numassign | stringassign | forstmt | nextstmt | colorstmt| inputstmt |

endstmt | clearstmt | refreshstmt | fastgraphicsstmt | dimstmt | pausestmt | arrayassign |

strarrayassign | openstmt | writestmt | closestmt| resetstmt

Uno statement può essere costituito da un'istruzione di salto (goto statement, gosub statement o return statement), da una funzione predefinita (print, plot, circle, rect, poly, line, color, input, cls, clg, refresh, fastgraphics, dim), da un'operazione su file (open, close, write, read, reset), da un'espressione di assegnamento (di variabili o di array), da un'istruzione di iterazione (for statement / next statememnt) o da istruzioni che agiscono sull'esecuzione (stop e pause statement).

gotostmt ::= GOTO variable

L'istruzione di salto goto modifica il normale flusso di esecuzione rimandandola alla label specificata dal non terminale variable.

gosubstmt ::= GOSUB variable

L'istruzione di salto gosub modifica il normale flusso di esecuzione rimandandola alla subroutine indicata dalla label specificata dal non terminale variable. L'esecuzione ritorna al flusso originario quando viene eseguita l'istruzione di return.

returnstmt ::= RETURN

L'istruzione di return consente di ripristinare il punto di esecuzione a partire da una subroutine invocata dall'istruzione gosub.

printstmt ::= PRINT stringexpr | PRINT (stringexpr) | PRINT floatexpr | PRINT

stringexpr ; | PRINT (stringexpr) ; | PRINT floatexpr ;

Il comando print crea una nuova riga e stampa il testo specificato dall'espressione (sia essa una stringexpr o una floatexpr) nella finestra di output destinata al testo. Se il punto e virgola opzionale è incluso, il comando stampa senza creare una nuova riga.

plotstmt ::= PLOT floatexpr , floatexpr | PLOT (floatexpr , floatexpr)

Il comando plot consente di settare al colore corrente il pixel della finestra di output destinata ai grafici, localizzato dalle coordinate specificate.

circlestmt ::= CIRCLE floatexpr , floatexpr , floatexpr | CIRCLE (floatexpr , floatexpr , floatexpr)

Il comando circle consente di disegnare col colore corrente un cerchio centrato nel punto di coordinate pari ai primi due argomenti e con raggio pari al terzo argomento.

rectstmt ::= RECT floatexpr , floatexpr , floatexpr , floatexpr | RECT (floatexpr , floatexpr , floatexpr , floatexpr)

Il comando rect consente di disegnare col colore corrente un rettangolo avente base e altezza pari rispettivamente ai primi due argomenti specificati; il vertice sinistro superiore è localizzato nel punto con coordinate pari agli ultimi due argomenti.

polystmt ::= POLY variable , floatexpr | POLY (variable, floatexpr) | POLY variable | POLY (variable) | POLY immediatelist

Il comando poly consente di disegnare un poligono. I lati del poligono sono definiti dai valori contenuti in un array (variable), che può essere precedentemente memorizzato o espresso come una lista di coppie di valori. Il numero di lati del poligono deve essere espresso con il secondo argomento.

linestmt ::= LINE floatexpr , floatexpr , floatexpr , floatexpr | LINE (floatexpr , floatexpr , floatexpr , floatexpr)

Il comando line consente di disegnare una linea che parte dal punto con coordinate pari alla prima coppia di argomenti e che termina nel punto le cui coordinate sono date dalla seconda coppia di argomenti.

numassign ::= variable = floatexpr

Assegnamento di una variabile numerica.

stringassign ::= stringvar = stringexpr

Assegnamento di una stringa.

forstmt ::= FOR variable = floatexpr TO floatexpr | FOR variable = floatexpr TO floatexpr STEP

floatexpr nextstmt ::= NEXT variable

Le istruzioni for e next vengono utilizzate unitamente per iterare l'esecuzione di un determinato gruppo di istruzioni. Alla prima esecuzione, la variabile viene settata al valore dell'espressione immediatamente a destra dell'operatore uguale. In seguito a ciascun comando next, la variabile viene incrementata di 1 (di default), oppure nel caso in cui venga utilizzata l'opzione step, del valore indicato dall'espressione alla destra di step. Le iterazioni proseguono finchè la variabile non raggiunge il valore dell'espressione alla destra del to.

colorstmt ::= SETCOLOR color | SETCOLOR (color)

Il comando setcolor consente di settare il colore del disegno a quello specificato.

inputstmt ::= inputexpr , stringvar | inputexpr , stringvar [floatexpr]

L'istruzione di input consente di inserire una linea di testo nella finestra di output del testo. Tale linea viene inoltre memorizzata in una stringvar.

endstmt ::= END

Comando con cui è possibile arrestare l'esecuzione.

clearstmt ::= CLS | CLG

Comando con cui è possibile pulire le finestre di output del testo e dei grafici rispettivamente.

refreshstmt ::= REFRESH

Con il comando refresh è possibile modificare la finestra di output dei grafici in modo che mostri tutti i disegni a partire del precedente comando refresh. Tale comando viene utilizzato in modalità fastgraphics.

fastgraphicsstmt ::= FASTGRAPHICS

Con tale comando è possibile attivare la modalità fastgraphics, fino all'arresto del programma. In modalità fastgraphics il grafico visualizzato non è modificabile fino a che non viene invocato il comando refresh.

dimstmt ::= DIM variable (floatexpr) | DIM stringvar (floatexpr)

Il comando dim consente la creazione di un array numerico di stringhe della dimensione specificata dalla floatexpr.

pausestmt ::= PAUSE floatexpr

Con il comando pause è possibile fermare l'esecuzione per il numero di secondi specificato.

arrayassign ::= variable [floatexpr] = floatexpr | variable = immediatelist

Assegnamento di un array numerico o di un elemento di un'array già esistente.

strarrayassign ::= stringvar [floatexpr] = stringexpr | stringvar = immediatestrlist

Assegnamento di un array di stringhe o di un elemento di un'array già esistente.

openstmt ::= OPEN (stringexpr) | OPEN stringexpr

Con il comando open è possibile aprire un file specificato.

writestmt ::= WRITE (stringexpr) | WRITE stringexpr

Con il comando write è possibile scrivere una stringa al termine del file corrente.

closestmt ::= CLOSE | CLOSE ( )

Con il comando close è possibile chiudere il file precedentemente aperto con il comando open.

resetstmt ::= RESET | RESET ( )

Con il comando reset è possibile cancellare tutti i dati contenuti nel file precedentemente aperto con il comando open.

ifexpr ::= IF compoundboolexpr

Valutazione dell'espressione booleana composta. Se essa risulta vera, lo statement presente nell'ifstmt viene eseguito, altrimenti viene ignorato.

compoundboolexpr ::= boolexpr| compoundboolexpr AND compoundboolexpr | compoundboolexpr OR compoundboolexpr | compoundboolexpr XOR compoundboolexpr | NOT compoundboolexpr % prec UMINUS | (compoundboolexpr)

Espressione booleana composta. Essa può essere costituita da un'espressione booleana semplice, oppure da un'operazione logica tra espressioni booleane composte.

boolexpr ::= stringexpr = stringexpr | stringexpr NE stringexpr | floatexpr = floatexpr | floatexpr NE floatexpr | floatexpr < floatexpr | floatexpr > floatexpr | floatexpr GTE floatexpr | floatexpr LTE floatexpr

Un'espressione booleana è composta da un'operazione logica o aritmetica tra costanti numeriche o da una relazione di uguaglianza (o non uguaglianza) tra stringhe.

floatexpr ::= (floatexpr) | floatexpr + floatexpr | floatexpr - floatexpr | floatexpr * floatexpr | floatexpr / floatexpr | floatexpr ^ floatexpr | - FLOAT %prec UMINUS | - INTEGER %prec UMINUS | - variable %prec UMINUS | FLOAT | INTEGER | KEY | variable [floatexpr] | variable | TOINT (floatexpr) | TOINT (stringexpr) | LENGTH (stringexpr) | INSTR (stringexpr , stringexpr) | CEIL (floatexpr) | FLOOR (floatexpr) | SIN (floatexpr) | COS (floatexpr) | TAN (floatexpr) | ABS (floatexpr) | RAND | PI

*Una floatexpr è una costante numerica (di tipo intero o float); essa può essere anche data dal risultato di un'operazione aritmetica tra costanti (+, -, *, /), dal risultato di un'operazione predefinita su interi o dalla conversione di stringhe in costante intera.*

stringexpr ::= stringexpr + stringexpr | STRING | stringvar [floatexpr] | stringvar | TOSTRING (floatexpr) | MID (stringexpr , floatexpr , floatexpr) | READ ( ) | READ

Con il non-terminale stringexpr si fa riferimento alle stringhe; esse possono essere anche ottenute come risultato di operazioni predefinite su stringhe o conversioni di costanti numeriche in stringhe.

immediatestrlist ::= { stringlist }

stringlist ::= stringexpr | stringexpr , stringlist

Con il non-terminale immediatestrlist si fa riferimento ad una lista composta da uno o più stringhe.

inputexpr ::= INPUT stringexpr | INPUT (stringexpr)

Come visto in precedenza (per il non-terminale inputstmt), l'istruzione di input consente di inserire una linea di testo nella finestra di output del testo.

immediatelist ::= { floatlist }

floatlist ::= floatexpr | floatexpr , floatlist

Con il non-terminale immediatelist si fa riferimento ad una lista composta da uno o più costanti numeriche.

stringvar ::= variable $

Le variabili che contengono stringhe seguono le stesse regole delle variabili contenenti valori numerici ma, a differenza di queste, terminano con il simbolo $.

label ::= variable :

Con il non terminale label si fa riferimento alle etichette utilizzate per indicare porzioni di codici. Esse sono costituite da una variabile seguita dal carattere due punti.

variable ::= letter { letter | digit }

Con il non-terminale variable si fa riferimento a una variabile. Le variabili contenenti valori numerici devono iniziare con una lettera e possono essere costituite da un qualsiasi numero di caratteri alfa-numerici. Le variabili sono case sensitive.

digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Con il non-terminale digit si fa riferimento a una qualsiasi cifra nell'intervallo [0-9]

letter ::= a | b | c | d | e | f | g | h | it | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z

| A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

Con il non-terminale letter si fa riferimento ad una qualsiasi lettera dell'alfabeto inglese, sia essa minuscola o maiuscola

BIBLIOGRAFIA

1. http://sisinflab.poliba.it/piscitelli, *dispense del corso di "Linguaggi formali e compilatori"*

2. http://www-ictserv.poliba.it/piscitelli/doc/lucidi%20LFC/bison%20GNU_pub.pdf, *manuale BISON del GNU*

3. http://flex.sourceforge.net/manual/, *manuale FLEX del GNU*

4. http://en.wikipedia.org/wiki/List_of_BASIC_dialects, *lista dei "dialetti" del Basic*

5. http://www.mat.uniroma2.it/~giammarr/Teaching/Compilatori/Lezioni/PrimaLezioneFlex.pdf

6. http://www.mat.uniroma2.it/~giammarr/Teaching/Compilatori/Lezioni/TerzaLezioneBison.pd

7. http://www.dii.unisi.it/~maggini/Teaching/TEL/slides/04%20-%20YACC.pdf

8. http://dinosaur.compilertools.net/bison/bison_6.html

9. http://thedance.net/kidbasic/KID/DOC/en/index.html

10. http://doc.basic256.org/doku.php?id=en:start, *Basic256 syntax*