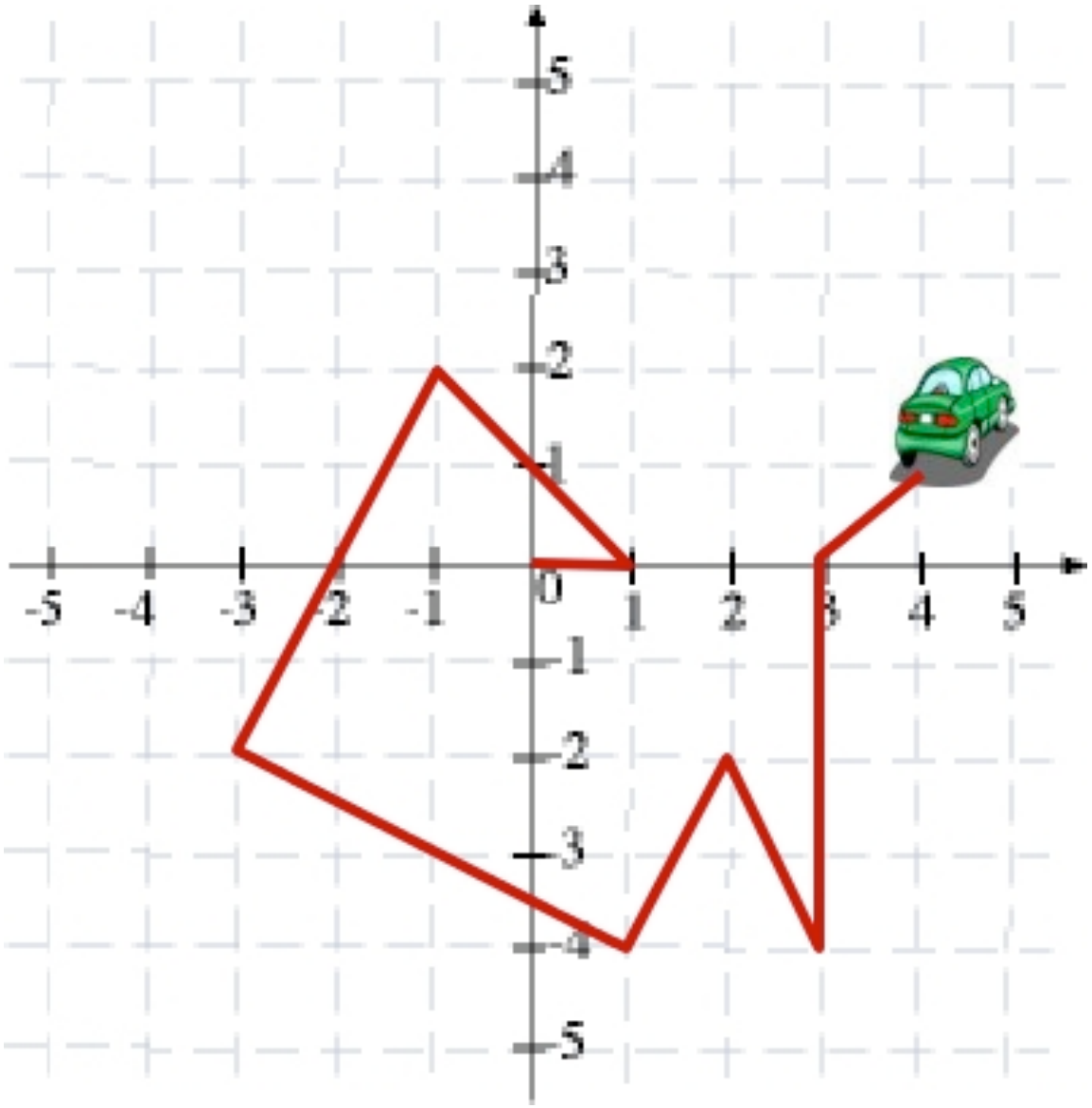
NAVIGATOR

TEORIE E TECNICHE DI COMPILAZIONE

a cura di *Gaspare Di Giovanna*
professoressa *Marinella Sciortino*



Gaspare Di Giovanna • E-mail: gdg_1981@yahoo.it • 23 novembre 2007

IL PROGETTO
COS'È E COME FUNZIONA

L'oggetto della tesina è un traduttore in grado di riconoscere un linguaggio che permette di descrivere il percorso di un veicolo all'interno di uno spazio bidimensionale.
Per la costruzione di Navigator ci avvarremo del LEX per quanto riguarda l'analisi lessicale e del BISON per l'analisi sintattica e semantica.
Utilizzeremo dei file di libreria in linguaggio C di supporto per effettuare ulteriori controlli (come l'analisi della validità della data) ed eseguire i comandi.

COMPILARE

Per compilare il progetto sotto Linux/Unix è possibile utilizzare il *Makefile* presente nella cartella:

`make`	`compila il progetto`
`make clean`	`cancella i file intermedi`

Sotto Windows è possibile utilizzare lo stesso *Makefile* se le dipendenze sono rispettate, in alternativa si possono utilizzare i due file batch:

`make.bat`	`compila il progetto`
`make_clean.bat`	`cancella i file intermedi`

ESEGUIRE

Per eseguire *navigator* sotto Linux/Unix la sintassi è la seguente:

```
./navigator < file_di input
```

mentre sotto Windows:

```
navigator.exe < file_di input
```

Sono presenti nella cartella *esempi* alcuni file di esempio da dare in pasto al nostro *navigator*.

COSTRUTTORI

Per quanto riguarda la parte dell'analisi sintattica utilizziamo delle espressioni regolari semplici per i costrutti più comuni:

spazi	[\t\n]
vuoti	{spazi}+
lettera	[a-zA-Z]
digit	[0-9]
reale	{digit}+("."{digit}+)?
data	{digit}{2}\/{digit}{2}\/{digit}{4}

Non avendo espressioni matematiche da valutare, utilizziamo lo stesso LEX per riconoscere e distinguere i reali positivi e negativi:

reale_neg	"-"{reale}
reale_pos	"+"?{reale}

tratteremo le frazioni come rapporto di numeri reali demandandone la valutazione al BISON.
La targa (il codice identificativo del veicolo) è costituita da una lettera maiuscola seguita da una stringa alfanumerica di 7 caratteri:

targa	[A-Z]({lettera}|{digit}){7}

Il commento è previsto su singola linea, racchiuso tra “<!” e “!>”

commento	\<![^\n]*!\>

Gli identificatori per le dimensioni di tempo/spazio:

metri	[mM]
secondi	[sS]

OPZIONI

Le uniche due opzioni presenti sono *noyywrap* per evitare di utilizzare l'opzione *-lfl* in fase di compilazione, e *yylineno* per identificare in fase di debugging la linea corrispondente all'errore rilevato durante il parsing dell'input.

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

AZIONI

Per togliere gli spazi basta l'azione nulla quando questi vengono incontrati:

{spazi}	;

Per la data, oltre a ritornare il relativo token, utilizziamo una funzione contenuta nella libreria di supporto che controlla la validità della data che riceve in input la stringa *yytext*:

{data}	{ check_date(yytext); return(DATA); }

Per i reali, positivi o negativi che siano, ritorniamo il valore dei relativi token, utilizzando la funzione interna di C *atof* che trasforma la stringa in reale:

{reale_neg}	{ yylval.real = atof(yytext); return(REALE_NEG); }
{reale_pos}	{ yylval.real = atof(yytext); return(REALE_POS); }

Per i lessemi che non prevedono associazione di valori o istruzioni ritorniamo semplicemente il relativo token:

{targa}	{ return(TARGA); }
{commento}	{ return(COMMENTO); }
Angle	{ return(ANGLE); }
Speed	{ return(SPEED); }
AcquireData	{ return(ACQUIREDATA); }
Move	{ return(MOVE); }
if	{ return(IF); }

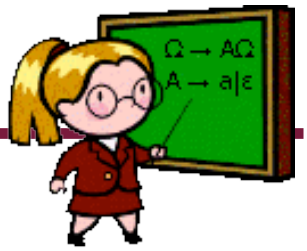
PI	{ return(PI); }
{metri}	{ return(M); }
{secondi}	{ return(S); }
"%%"	{ return('%'); }

Gli operatori relazionali, lo slash, le parentesi e il punto e virgola vengono restituiti al BISON tali e quali tramite l'azione comune *return(yytext[0])*:

"<"	|
">"	|
"/"	|
"["	|
"]"	|
"("	|
")"	|
";"	{ return(yytext[0]); }

L'errore è costituito logicamente da tutto ciò che non è previsto essere un lessema per il nostro linguaggio, è riconosciuto dal simbolo "." e identificato per il debugging tramite *yylineno* che ne segnala la riga corrispondente:

.	{ printf("ERRORE in riga %d!\n",yylineno);}



DICHIARAZIONI C

Le dichiarazioni C in testa al file *parser.y* prevedono l'inclusione della libreria C, la variabile *if_bool* utilizzata per l'utilizzo del costrutto *if*, la variabile *error* che conta il numero di errori rilevati e *veicolo*, una struttura contenente tutte le informazioni relative alla posizione, velocità, etc.

```
%{
#include "libreria.h"
int if_bool = 1;
int error;
t_car veicolo = {0,0,0,0,0,0};
%}
```

DICHIARAZIONI BISON

Come prima parte delle dichiarazione di BISON troviamo

```
%union{
  float real;
  char flag;
}
```

che indica la collezione dei possibili tipi di dato per i valori semantici.
Nel nostro caso abbiamo il tipo *float* per i valori numerici e il tipo *char* che serve per identificare le flag per le dimensioni spazio/tempo e per gli operatori relazionali.

Subito dopo dichiariamo i token

```
%token TARGA COMMENTO DATA
%token SPEED ANGLE MOVE ACQUIREDATA IF
%token PI M S
```

i token tipati

```
%token <real> REALE_POS REALE_NEG
```

e i tipi per i simboli non terminali della grammatica che utilizziamo

```
%type  <real> reale angolo
%type  <flag> spazio_tempo gradi_radianti relop
```

REGOLE GRAMMATICALI

La regola base dice che il file deve iniziare con la *TARGA*, seguita dalla *DATA*, la regola *comment*, il token **%** (ricavato da %%) e la regola *first_statement*

```
root
        : TARGA DATA comment '%' first_statement
        ;
```

Il commento può essere presente come può non esserci

```
comment
        : /* empty */
        | COMMENTO
        ;
```

Abbiamo previsto che il file potesse finire qua (nessun comando), altrimenti deve obbligatoriamente cominciare con la regola che setta la velocità *speed_action*, seguita dall'elenco delle possibili istruzioni *statements*

```
first_statement
        : /* empty */
        | speed_action statements
        ;
```

Le istruzioni sono di tre tipi: l'istruzione vuota, l'*if_case* nel caso in cui abbiamo una condizione e le *actions*, questi ultimi due seguiti da *statements*

```
statements
        : /* empty */
        | if_case statements
        | actions statements
        ;
```

i due tipi di istruzione servono per garantire che non vi siano *if* annidati, una volta entrati nell'*if_case*, infatti, a condizione vera possiamo eseguire soltanto azioni all'interno delle parentesi quadre (non possiamo ritornare in *statements*)

```
if_case
        : IF '(' condition ')' '[' if_statements ']'
        ;
if_statements
        : /* empty */
        | actions if_statements
        ;
actions
        : speed_action
        | move_action
        | angle_action
        | acquire_action speed_action
        ;
```

La condizione prevede che ci sia un identificatore *spazio_tempo* per stabilire se dobbiamo confrontare il tempo o lo spazio trascorsi, *relop* per l'operatore del confronto e il valore *reale* per il secondo termine del confronto

```
condition
        : spazio_tempo relop reale
        ;
spazio_tempo
        : M
        | S
        ;
relop
        : '<'
        | '>'
        ;
reale
        : REALE_POS
        | REALE_NEG
        ;
```

Abbiamo quindi le azioni che possiamo far eseguire al nostro veicolo tramite gli appositi comandi, ogni azione accetta ovviamente i suoi parametri.
I più "complessi" sono *move_action* che ha bisogno di *spazio_tempo* per stabilire la dimensione dalla quale dipenderà il movimento e *angle_action* che può avere come parametri l'angolo in gradi o in radianti

```
speed_action
        : SPEED '(' REALE_POS ')' ';'
        ;
move_action
        : MOVE '(' REALE_POS spazio_tempo ')' ';'
        ;
angle_action
        : ANGLE '(' angolo gradi_radianti ')' ';'
        ;
acquire_action
        : ACQUIREDATA '(' REALE_POS ')' ';'
        ;
```

L'*angolo* dell'azione *set_angle* può essere un reale o una frazione, ma al nostro fine è sembrata una buona scelta farci comunque restituire dal BISON un valore di tipo float.
Il flag *gradi_radianti* produce il simbolo ε oppure *PI* nel caso in cui l'angolo sia in radianti.

```
angolo
        : reale
        | reale '/' REALE_POS
        ;
gradi_radianti
        : /* empty */
        | PI
        ;
```

CODICE C ADDIZIONALE

Il codice addizionale presente nell'ultima parte del file di BISON contiene la funzione *yyerror* che gestisce l'output degli errori e ne conta le occorrenze

```
int yyerror(char* s)
{
  fprintf(stderr,"%s\n",s);
  error++;
}
```

e la dichiarazione del main con la funzione *yyparse()* che richiama la routine di parsing del file di input

```
int main(void)
{
  yyparse();
  return 0;
}
```

FILES C DI SUPPORTO

libreria.c, libreria.h

HEADER

Il file *libreria.h* contiene le librerie utilizzate

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

la definizione della struttura *t_car* che contiene tutti i dati relativi al veicolo

```
typedef struct s_car{
  float time;
  float space;
  float speed;
  float angle;
  float posX;
  float posY;
} t_car;
```

e i prototipi delle funzioni utilizzati nel file *libreria.c*

```
void check_date  (char *date);
int  check_cond  (t_car *veicolo,char flag,char relop,float real);

void SetSpeed    (t_car *veicolo,float new_speed,           int if_bool);
void SetAngle    (t_car *veicolo,float new_angle, char flag,int if_bool);
void MoveVehicle (t_car *veicolo,float space_time,char flag,int if_bool);
void AcquireData (t_car *veicolo,float time,               int if_bool);

void Report      (t_car* veicolo, int error);
```

FUNZIONI

La prima funzione che incontriamo è quella del controllo sulla data

```
void check_date(char *date)
{
  int gg,mm,aaaa;

  /* trasformiamo la stringa in numeri */
  gg   = atoi((char*)(&date[0]));
  mm   = atoi((char*)(&date[3]));
  aaaa = atoi((char*)(&date[6]));
  /* effettuiamo il controllo sulla validità della data */
  { ... }
  /* se la data non è valida richiamiamo la funzione yyerror */
  yyerror("ERRORE: Data non valida!");
}
```

quindi abbiamo la funzione che setta la variabile *if_bool* a 0 se *condition* dell'*if_case* non è soddisfatta

```
if_case
        : IF '(' condition ')' '[' if_statements ']'
        ;
```

in questa maniera ciò che è contenuto all'interno delle parentesi quadre viene parsato ma non eseguito e *if_bool* viene riportato a 1 dopo la chiusura di queste

```
int check_cond(t_car *veicolo, char flag, char relop, float valore)
{
  /* controlliamo se il confronto deve essere fatto con il tempo
     o con lo spazio trascorsi                                    */
  float control_value = (flag == 's')?veicolo->time:veicolo->space;
  /* ritorniamo 1 se la condizione è vera o 0 se è falsa          */
  return((relop == '<')?(control_value<valore):(control_value>valore));
}
```

Tutte le azioni ricevono il parametro *if_bool*, dato che possono essere eseguite anche all'interno dell'*if_case*, ed effettuano subito il controllo su questo

```
void SetSpeed(t_car *veicolo, float new_speed, int if_bool)
{
  if(if_bool)
  {
    /* aggiorniamo la velocità del veicolo e incrementiamo il tempo */
    veicolo->speed = new_speed;
    (veicolo->time)++;
  }
}
```

In *SetAngle*, come nelle altre funzioni dov'è presente, viene fatto il controllo su un unico valore di *flag*, dato che comunque possono sempre esserci 2 soli casi

```
void SetAngle(t_car *veicolo, float new_angle, char flag, int if_bool)
{
  if(if_bool)
  {
    /* controlliamo la flag per sapere se l'angolo è in gradi o radianti */
    if(flag == 'g')
      /* nel primo caso riconduciamo il valore in radianti */
      new_angle /= 180;
    /* aggiorniamo la direzione del veicolo e incrementiamo il tempo */
    veicolo->angle = new_angle*PI;
    (veicolo->time)++;
  }
}
```

```
void MoveVehicle(t_car *veicolo, float space_time, char flag, int if_bool)
{
  if(if_bool)
  {
    if(veicolo->speed != 0)
    {
      /* controlliamo se siamo nel caso temporale o spaziale */
      if(flag == 's')
      /* nel primo caso riconduciamo la variabile in spaziale */
        space_time *= veicolo->speed;

      /* aggiorniamo il valore dello spostamento */
      (veicolo->space) += space_time;
      /* aggiorniamo la posizione del veicolo */
      (veicolo->posX)  += space_time * (cos(veicolo->angle));
      (veicolo->posY)  += space_time * (sin(veicolo->angle));

      /* aggiorniamo il valore del tempo trascorso */
      (veicolo->time)  += (space_time / veicolo->speed);
    }
  }
}
```

```
void AcquireData(t_car *veicolo, float time, int if_bool)
{
  if(if_bool)
    /* incrementiamo il tempo del viaggio */
    (veicolo->time) += time;
}
```

Come ultima funzione abbiamo quella che genera l'output, questa effettua il controllo sul numero di errori (che vedremo comunque rilevati prima di arrivare a questo punto) e, nel caso in cui non vi siano errori, restituisce anche la posizione finale del veicolo e il tempo trascorso

```
void Report(t_car* veicolo, int error)
{
  if(error)
    /* se è stato contato almeno un errore: */
    fprintf(stdout,"Sintassi non corretta \n");
  else
  {
    /* altrimenti restituiamo l'output richiesto: */
    fprintf(stdout,"Sintassi corretta\n");
     fprintf(stdout,"Posizione finale: ...",veicolo->posX,veicolo->posY);
     fprintf(stdout,"Tempo trascorso: %.2f sec\n",veicolo->time);
  }
}
```