# Linguaggi e computabilità

Laboratorio 3: compilare il linguaggio ‘Simple’

- Compilare un linguaggio giocattolo: ‘Simple’
  - Approccio e architettura
  - Un programma esempio
  - Linguaggio ‘Simple’
  - Ipotesi di macchina e Istruzioni assembly
  - Struttura lexer e parser
  - Proposte di scrittura programmi in ‘Simple’
  - Esercizi di modifica del linguaggio

# Approccio e architettura

- Scriviamo programmi con un (piccolo)
  linguaggio di alto livello

- Il lexer estrae i token e li passa al parser
- Il parser li traduce in un output assembly
  per una macchina immaginaria (a stack)

- Opzionale: l'assembly viene eseguito
  con un (piccolo) simulatore
  [in 'C', da compilare: SM.h + stackMachine.c ]

# Esempio (1/3)

```
// calcolare il quadrato come ciclo di somme,
// n*n = n+n+...+n, n volte sommato
// R1: numero n (<10) di cui stampare il quadrato
// R2: contatore per il ciclo di somme
// R3: risultato, da stampare
```

# Esempio (2/3)

```
BEGIN
  read R1; // chiedo n
  if R1 < 10 then
    // […ciclo di somme in R3, slide seguente...]
    // [...stampa R3, slide seguente…]
  else
    skip; // non calcolo nulla
  fi;
END
```

# Esempio (3/3)

```
R2 := R1 ; // numero di somme *mancanti*
                // ... inizialmente n
R3 := 0 ; // vi calcolo il risultato
while R2 > 0 do
  R3 := R3+R1;
  R2 := R2-1;
done ;

write R1; write R3;
```

# Il linguaggio ‘Simple’

- program ::= ‘BEGIN’ command_s ‘END’
- identifier ::= ‘R0’ | ‘R1’ | … | ‘R9’ , comment::= ‘//’...riga
- command_s ::=  | command_s command ‘;’
- command ::=  ‘skip’
  | ‘read’ identifier
  | ‘write’ exp
  | identifier ‘:=’ exp
  | ‘if’ exp ‘then’ command_s ‘else’ command_s ‘fi’
  | ‘while’ exp ‘do’ command_s ‘done’
- exp ::= NUMBER | identifier | ’(’ exp ’)’
  | exp ‘+’ exp | exp ‘−’ exp | exp ‘*’ exp | exp ‘/’ exp
  | exp ‘=’ exp | exp ‘<’ exp | exp ‘>’ exp

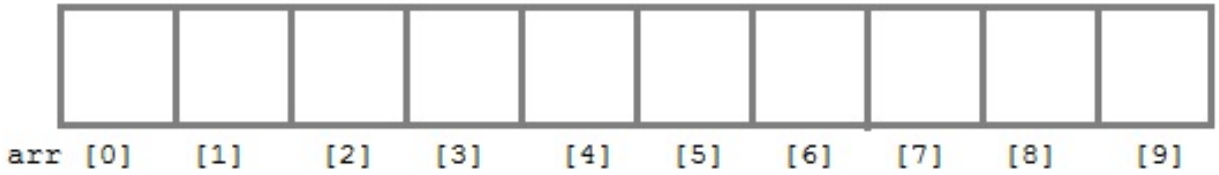
# Macchina e relativo assembly

Macchina immaginaria, a **stack** (inizia vuoto: top= -1):

- 10 variabili (R0...R9) in inizio stack (alloco con ‘DATA 10’)
- Operazioni prendono 2 valori ‘top’ di stack
  e mettono risultato in top
- ‘false’ e ‘true’ sono i valori numerici ‘0’ e ‘1’
- Ho 4 operazioni aritmetiche e 3 confronti
- Su top: metto con ‘LOAD_INT num’ e ‘LOAD_VAR n’,
  tolgo con ‘STORE n’ (da top stack a variabile n)
- Ho ‘GOTO addr’, ‘JMP_FALSE addr’, ‘HALT’
- Ho I/O: ‘READ_INT’, numero da tastiera → top
  ‘WRITE_INT’, numero da top → schermo

# Struttura dello stack

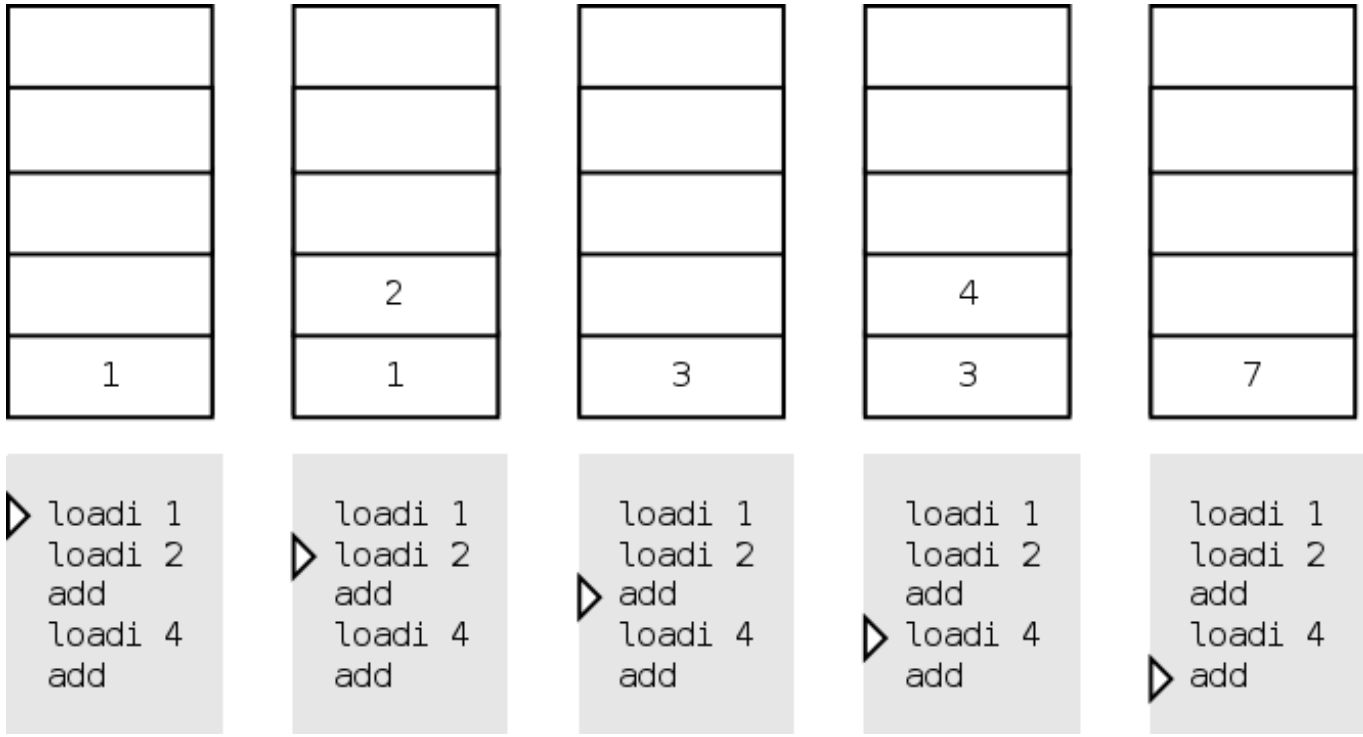
- Spazio per R0..R9:
  (è‘DATA 10’ iniziale)




Top
(dopo DATA 10)

- Esempio di esecuzione: (1+2)+4

  due somme tra costanti (che vengono messe su stack)



# Struttura lexer

Il lexer produce un token per ogni:

- Parola chiave (come ‘done’)
- Operatore di assegnamento ‘:=’
- Numero (consegna valore in yyparser.yylval )
- Identificatore (consegna indice della variabile, ad esempio 1 per ‘R1’, in yyparser.yylval )

Inoltre:

- Elimina spazi e commenti (‘//’, fino a fine riga)
- Passa gli altri caratteri direttamente (‘+’, ‘-’, ...)

# Frammento del lexer

```
%%
ID=R[0-9]                          (Nota: separo 'R' dalla cifra, poi...)
%%
[...]
write { return(Parser.WRITE); }
{ID} { yyparser.yylval = new     (...poi qui uso la cifra come id.)
    ParserVal(Integer.parseInt(yytext().substring(1)));
    return(Parser.IDENTIFIER); }
[ \t\n]+ { }
"//".* { }
[^] { return yytext().charAt(0); }
```

# Struttura parser

- I token NUMBER e IDENTIFIER hanno valore intero (il secondo per l'indice d'ogni variabile)
- I token IF e WHILE hanno valore intero per necessità di trattamento degli indirizzi di salto: non analizzeremo questo meccanismo
- Le istruzioni assembly sono elencate con

  public enum I { HALT, STORE, […] }
- Le istruzioni assembly sono generate con:

  gen_code( I.HALT, […]) [ad esempio]
- La parte 'if...then' della struttura 'if...then...else' ha un suo non terminale e una sua produzione (richiamata dalla produzione associata a 'if...then...else')

# Frammenti del parser

- Per ‘if...then’:

  ifThen : IF exp { $1 = [calcolo indir.jump] }

  THEN commands { $$ = $1; } ;

- Si noti che le azioni sono a volte ‘sparse’ lungo le produz.

- Alcuni comandi: command : SKIP [nessuna istr.ass.]

  | READ IDENTIFIER { gen_code( I.READ_INT, -99 );

  gen_code( I.STORE, $2 ); } [due istr.ass.]

  | WRITE exp { gen_code( I.WRITE_INT, -99 ); }

  | IDENTIFIER ASSGNOP exp { gen_code( I.STORE, $1 ); }

  (NB: le istr.senza arg., come READ_INT, sono generate con arg.-99)

- I non terminali ‘exp’ generano istruzioni assembly che lasciano il risultato in top stack

- Si noti l’uso dei valori di ‘IDENTIFIER’ ($2 in READ, $1 in assegn.)

# Output del parser

- Input (da dare da tastiera, o da redirezione ‘<file’) in linguaggio Simple:

```
BEGIN
  if R1<0 then R3:=100-R1;
  else skip; fi;
END
```

- Output (formato: indirizzo – codiceIstr / mnemonicoIstr arg ):

```
0-4/DATA 10                    // è BEGIN

1-6/LD_VAR 1    2-5/LD_INT 0    3-9/LT -99    4-2/JMP_FALSE 10 // if R1<0

5-5/LD_INT 100    6-6/LD_VAR 1    7-13/SUB -99    8-1/STORE 3 // è then ...
9-3/GOTO 10 // fine then (NB: else è vuoto ...)

10-0/HALT 0 // è END
```

# Esercizi da svolgere in laboratorio

Programmi proposti:

- Chiede 3 numeri, e stampa il più grande
- Chiede n, stampa i primi n numeri pari
- Chiede n, stampa fibonacci(n):

  fibonacci(n)= 0 se n=0,

  altrimenti è: fibonacci(n-2)+fibonacci(n-1)

- ...

# Esercizi per laboratorio e consegna

- Modifiche al linguaggio proposte:

- Aggiungere costanti ‘false’ e ‘true’ (val. 0 e 1)
- Aggiungere operazione ‘and’, ma N.B.:
  **tra sole espressioni con ‘false’, ‘true’, ‘and’**
  (cost_b::=‘false’|’true’ , espr_b::=espr_b ‘and’ cost_b …)
  (Nota: op.‘and’ tra valori 0 e 1 è la semplice moltipl.aritm.)
- Opzionale: Aggiungere ‘if’ senza ‘else’ (solo ‘if...then...fi’)
- Aggiungere assegnamento duale:  dual::=id id ‘DUAL’ exp
  (il valore di exp viene assegnato a entrambi gli ident.)