Compilare il linguaggio ‘Simple’

Linguaggi e computabilità

Terzo Laboratorio

- Approccio e architettura
- Un programma esempio
- Presentazione del 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 simulatore in 'C' [da compilare: SM.h + stackMachine.c]

Esempio

// calcolare il quadrato di un intero (n < 10)
//come ciclo di somme,
// n*n = n+n+...+n (n sommato n volte)

// R1: numero intero n (<10) di cui si vuole //stampare il quadrato
// R2: contatore per il ciclo di somme
// R3: risultato, da stampare

```
BEGIN
// leggo n
read R1;
if R1 < 10 then
// [...ciclo di somme in R3]
R2 := R1 ; // contatore somme *mancanti*
            // ... inizialmente n
R3 := 0 ; // risultato, da stampare alla fine
while R2 > 0 do
R3 := R3+R1;
R2 := R2-1;
done ;
// [stampa R1 e R3]
write R1; write R3;
else
// non calcolo nulla
skip;
fi;
END
```

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 (allocate con ‘DATA 10’)
- ‘false’ e ‘true’ corrispondono ai valori numerici ‘0’ e ‘1’
- Abbiamo a disposizione 4 operazioni aritmetiche e 3 di confronto
- Tutte le operazioni prendono 2 valori dal ‘top’ dello stack e mettono il risultato in top
- Su top:
  - metto con ‘LOAD_INT num’ e ‘LOAD_VAR n’
  - tolgo con ‘STORE n’ (da top stack a variabile n)
- Abbiamo a disposizione
  - ‘GOTO addr’, ‘JMP_FALSE addr’, ‘HALT’
  - I/O: ‘READ_INT’, numero letto da tastiera viene messo in ‘top’ di stack
  - ‘WRITE_INT’, numero da ‘top’ di stack viene scritto a schermo

Struttura dello stack

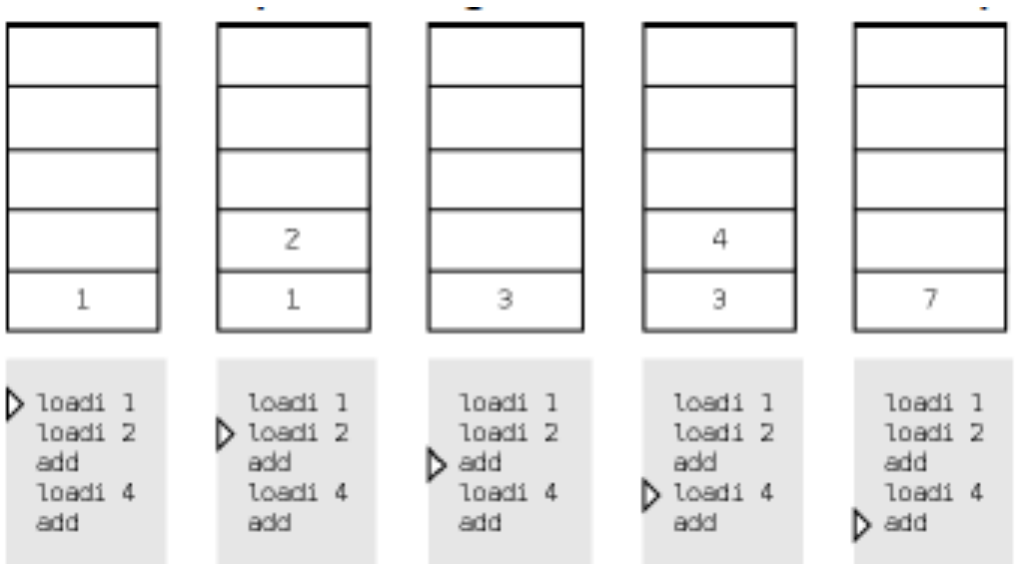
- Spazio per R0..R9:

(è ‘DATA 10’ iniziale)

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

due somme tra costanti (che vengono messe su stack)

Top
(dopo DATA 10)



Struttura del lexer

Il lexer produce un token per ogni:

- Parola chiave (e.g. ‘done’)
- Operatore di assegnamento ‘:=’
- Numero (consegna valore in yyparser.yylval)
- Identificatore (consegna l’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 di 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 produzioni
- 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 istruzioni senza argomento, come READ_INT, sono generate con argomento -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 assegnamento)

Output del parser

- Input (da dare da tastiera, o da redirezione ‘<file’) in linguaggio Simple
- Output (formato: indirizzo – codice Istruzione / mnemonico Istruzione arg)

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

```
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 ...
                            //(NB: else è vuoto ...)
9-3/GOTO 10                 // fine then
10-0/HALT 0                 // è END
```

How-to

- Generare il parser (come al solito con jflex e yacc)
- Scrivere un programma simple in un file (es. programma.s)
- Invocare "java Parser < programma.s" e salvare il contenuto in un file (es. compilato.txt)
- Eseguire la stackMachine con il file assembly ottenuto, invocando da terminale "stackMachine.exe compilato.txt"

- Si dovrà prima compilare il sorgente stackMachine.c con gcc

Esercizi laboratorio

Esercizio 1 – Provare il funzionamento di lexer e parser per Simple

- Scrivere con il linguaggio Simple i seguenti programmi
  - Chiede 3 numeri, e stampa il più grande
  - Chiede un intero n, stampa i primi n numeri pari

Esercizio 2 – Modifica del linguaggio Simple

- Aggiungere le costanti ‘false’ e ‘true’ (val. 0 e 1)
- Aggiungere l’operazione ‘and’ MA 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 moltiplicazione aritmetica)

- Aggiungere l’istruzione ‘if’ senza ‘else’ (ovvero, solo ‘if...then...fi’)
- Aggiungere assegnamento duale: dual::=id id ‘DUAL’ exp

(il valore di exp viene assegnato a entrambi gli identificatori)