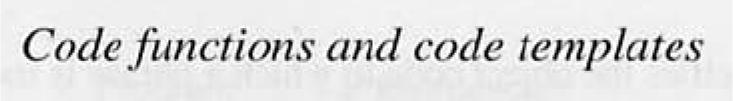
Geração de código

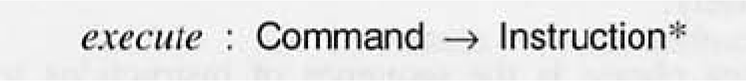
Baseado no Capítulo 6 de Programming Language Processors in Java, de Watt & Brown

Table C.2 Summary of TAM instructions.

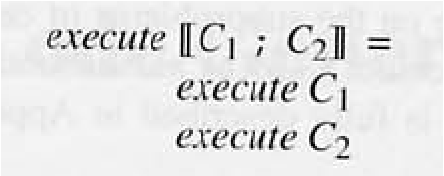
Op-code	Instruction mnemonic	Effect
0	LOAD(*n*) *d*[*r*]	Fetch an *n*-word object from the data address (*d* + register *r*), and push it on to the stack.
1	LOADA *d*[*r*]	Push the data address (*d* + register *r*) on to the stack.
2	LOADI(*n*)	Pop a data address from the stack, fetch an *n*-word object from that address, and push it on to the stack.
3	LOADL *d*	Push the 1-word literal value *d* on to the stack.
4	STORE(*n*) *d*[*r*]	Pop an *n*-word object from the stack, and store it at the data address (*d* + register *r*).
5	STOREI(*n*)	Pop an address from the stack, then pop an *n*-word object from the stack and store it at that address.
6	CALL(*n*) *d*[*r*]	Call the routine at code address (*d* + register *r*), using the address in register *n* as the static link.
7	CALLI	Pop a closure (static link and code address) from the stack, then call the routine at that code address.
8	RETURN(*n*) *d*	Return from the current routine: pop an *n*-word result from the stack, then pop the topmost frame, then pop *d* words of arguments, then push the result back on to the stack.
9	–	(unused)
10	PUSH *d*	Push *d* words (uninitialized) on to the stack.
11	POP(*n*) *d*	Pop an *n*-word result from the stack, then pop *d* more words, then push the result back on to the stack.
12	JUMP *d*[*r*]	Jump to code address (*d* + register *r*).
13	JUMPI	Pop a code address from the stack, then jump to that address.
14	JUMPIF(*n*) *d*[*r*]	Pop a 1-word value from the stack, then jump to code address (*d* + register *r*) if and only if that value equals *n*.
15	HALT	Stop execution of the program.



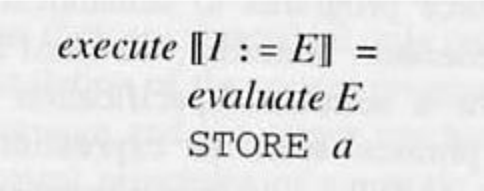
Code functions and code templates



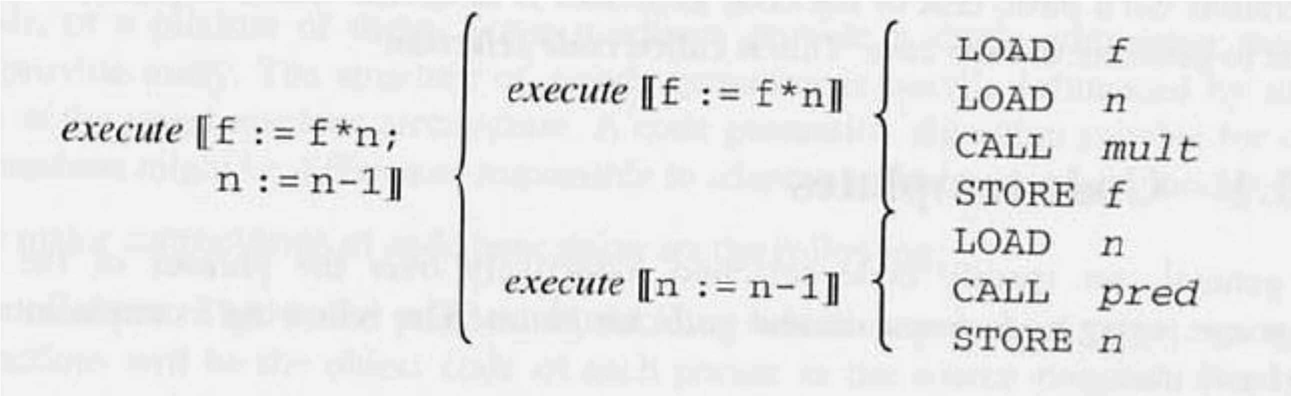
$$execute : \text{Command} \rightarrow \text{Instruction}^*$$



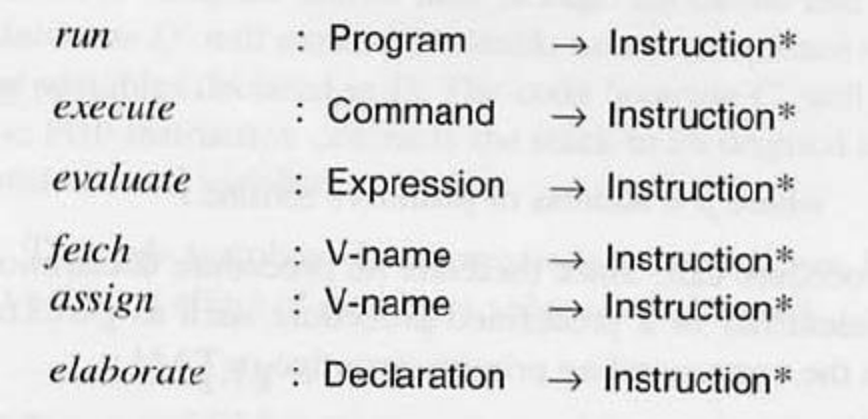
$$\begin{aligned} execute\ [\![C_1\ ;\ C_2]\!] = \\ & execute\ C_1 \\ & execute\ C_2 \end{aligned}$$



$$
\begin{array}{l}
\textit{execute}\,[\![I\ :=\ E]\!]\ = \\
\qquad \textit{evaluate}\ E \\
\qquad \texttt{STORE}\ \ a
\end{array}
$$



$$
\begin{array}{l}
\textit{execute}\,[\![\texttt{f := f*n;} \\
\qquad\quad \texttt{n := n-1}]\!]
\end{array}
\left\{
\begin{array}{l}
\textit{execute}\,[\![\texttt{f := f*n}]\!]
\left\{
\begin{array}{ll}
\texttt{LOAD} & f \\
\texttt{LOAD} & n \\
\texttt{CALL} & \textit{mult} \\
\texttt{STORE} & f
\end{array}
\right. \\
\textit{execute}\,[\![\texttt{n := n-1}]\!]
\left\{
\begin{array}{ll}
\texttt{LOAD} & n \\
\texttt{CALL} & \textit{pred} \\
\texttt{STORE} & n
\end{array}
\right.
\end{array}
\right.
$$



run	:	Program	→	Instruction*
execute	:	Command	→	Instruction*
evaluate	:	Expression	→	Instruction*
fetch	:	V-name	→	Instruction*
assign	:	V-name	→	Instruction*
elaborate	:	Declaration	→	Instruction*

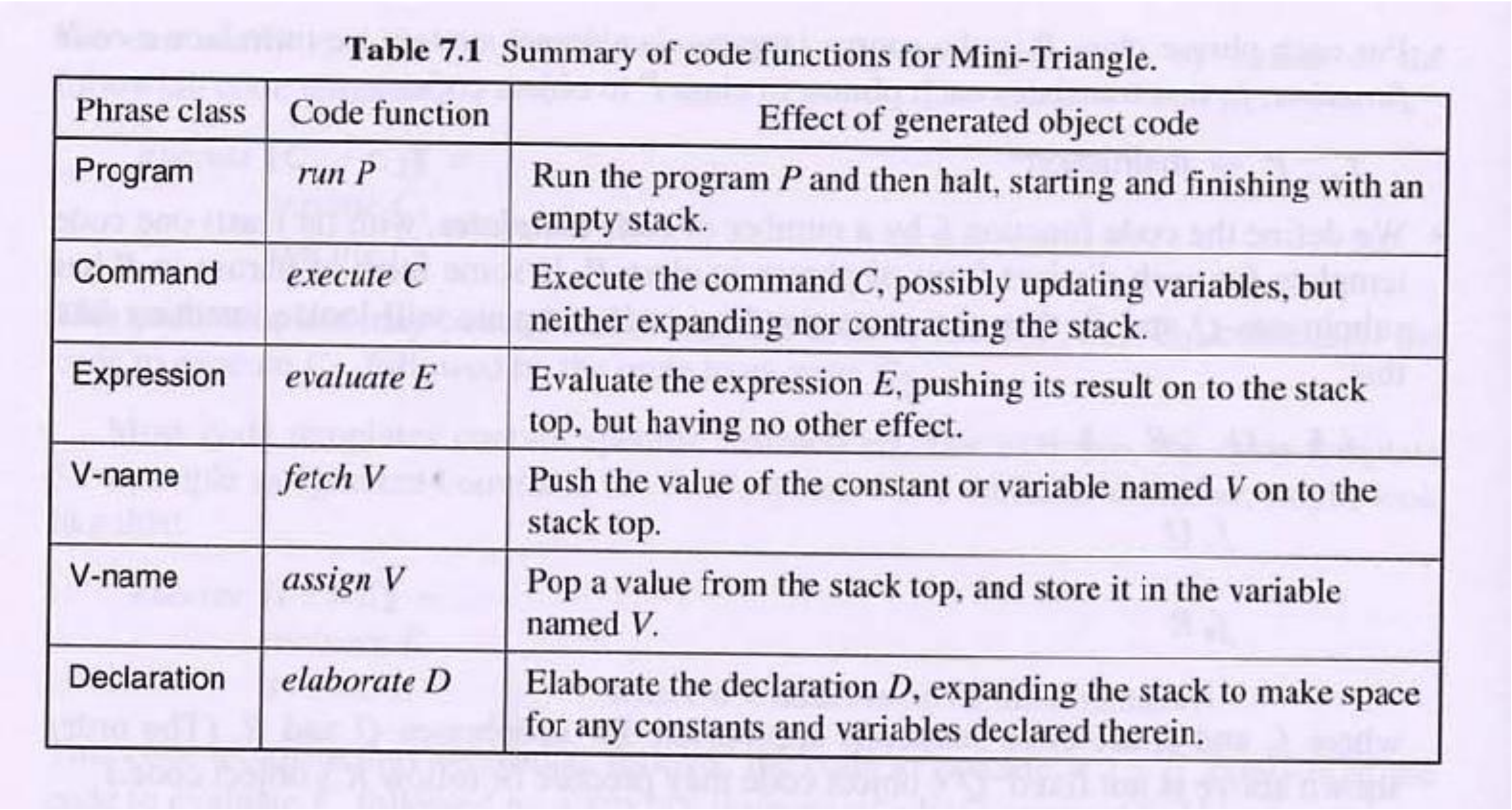
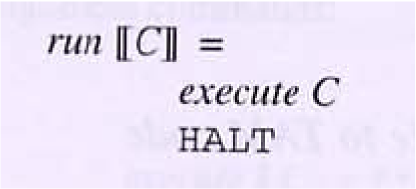
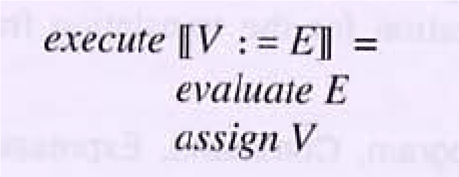


Table 7.1 Summary of code functions for Mini-Triangle.

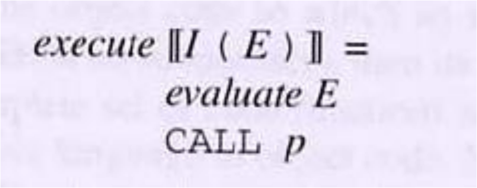
Phrase class	Code function	Effect of generated object code
Program	*run P*	Run the program *P* and then halt, starting and finishing with an empty stack.
Command	*execute C*	Execute the command *C*, possibly updating variables, but neither expanding nor contracting the stack.
Expression	*evaluate E*	Evaluate the expression *E*, pushing its result on to the stack top, but having no other effect.
V-name	*fetch V*	Push the value of the constant or variable named *V* on to the stack top.
V-name	*assign V*	Pop a value from the stack top, and store it in the variable named *V*.
Declaration	*elaborate D*	Elaborate the declaration *D*, expanding the stack to make space for any constants and variables declared therein.



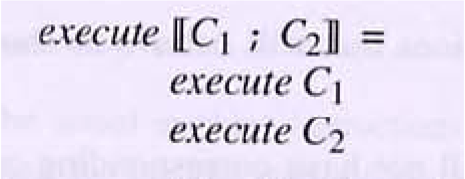
```
run ⟦C⟧ =
        execute C
        HALT
```



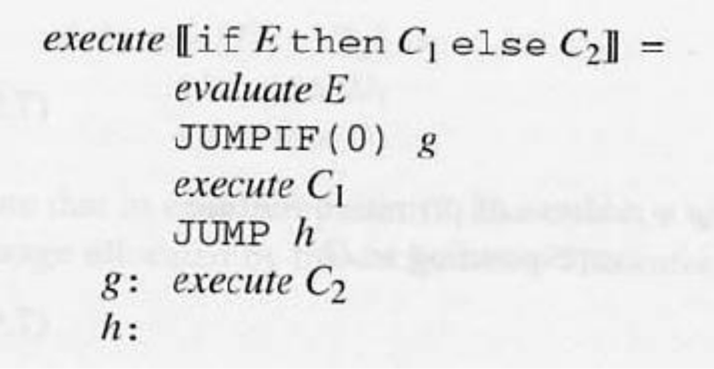
```
execute ⟦V := E⟧ =
        evaluate E
        assign V
```



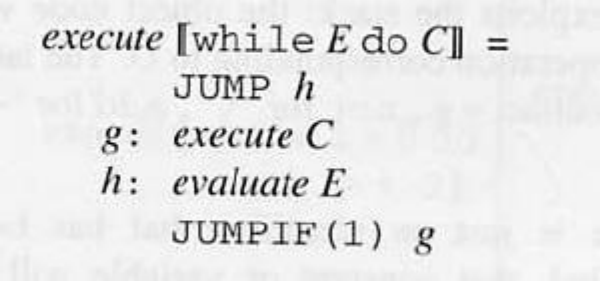
```
execute ⟦I ( E )⟧ =
        evaluate E
        CALL p
```



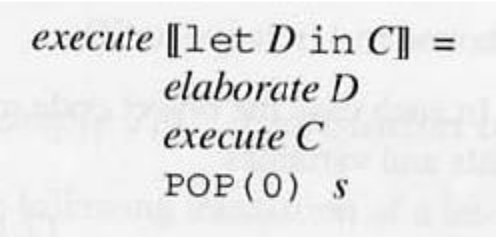
```
execute ⟦$C_1$ ; $C_2$⟧ =
        execute $C_1$
        execute $C_2$
```



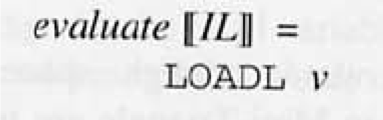
```
execute ⟦if E then $C_1$ else $C_2$⟧ =
          evaluate E
          JUMPIF(0)  g
          execute $C_1$
          JUMP  h
     g:   execute $C_2$
     h:
```



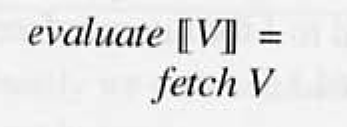
```
execute ⟦while E do C⟧ =
          JUMP  h
     g:   execute C
     h:   evaluate E
          JUMPIF(1)  g
```



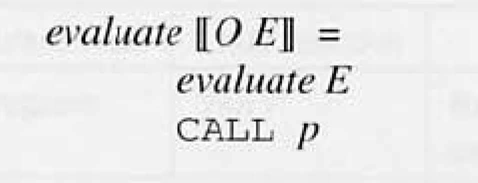
```
execute ⟦let D in C⟧ =
        elaborate D
        execute C
        POP(0) s
```



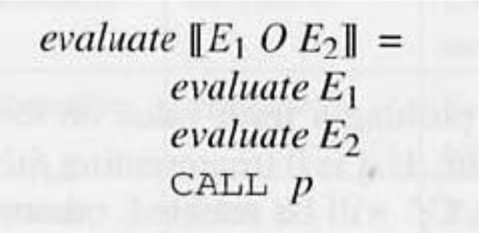
```
evaluate ⟦IL⟧ =
        LOADL v
```



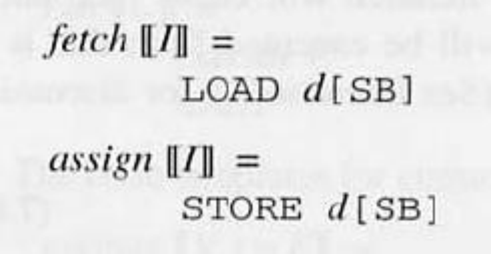
```
evaluate ⟦V⟧ =
        fetch V
```



```
evaluate ⟦O E⟧ =
        evaluate E
        CALL p
```

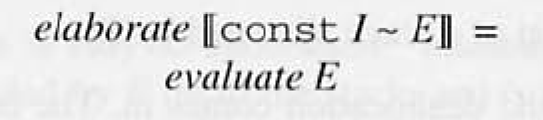


```
evaluate ⟦$E_1$ O $E_2$⟧ =
        evaluate $E_1$
        evaluate $E_2$
        CALL p
```

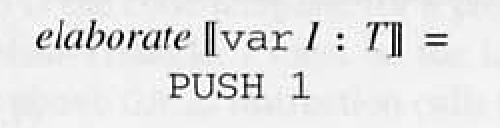


```
fetch ⟦I⟧ =
        LOAD d[SB]

assign ⟦I⟧ =
        STORE d[SB]
```

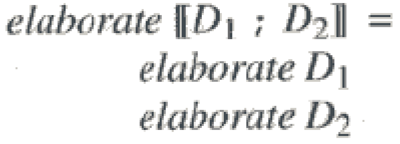


$$\mathit{elaborate}\ [\![\texttt{const}\ I \sim E]\!] =$$
$$\mathit{evaluate}\ E$$



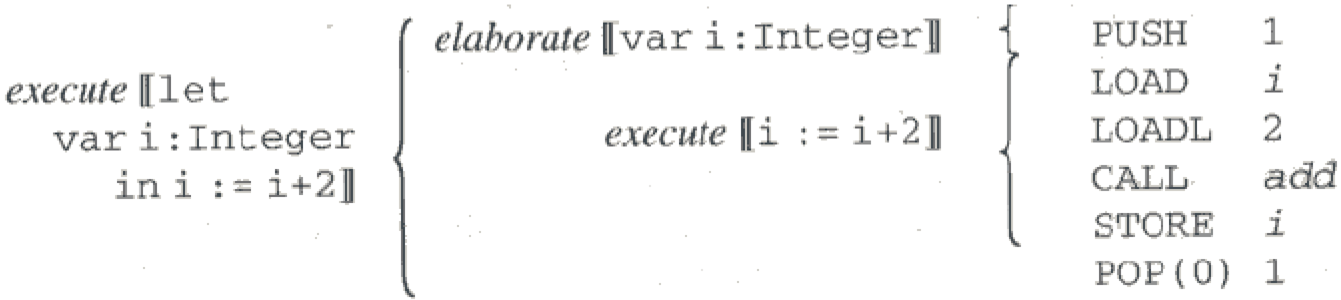
$$\mathit{elaborate}\ [\![\texttt{var}\ I : T]\!] =$$

```
PUSH 1
```



$$\mathit{elaborate}\ [\![D_1\ ;\ D_2]\!] =$$
$$\mathit{elaborate}\ D_1$$
$$\mathit{elaborate}\ D_2$$

```
                                                        30:  JUMP   35
                                                        31:  LOAD   i
                            execute ⟦i := i-2⟧          32:  LOADL  2
execute ⟦while i > 0 do                                 33:  CALL   sub
              i := i-2⟧                                 34:  STORE  i
                                                        35:  LOAD   i
                              evaluate ⟦i > 0⟧          36:  LOADL  0
                                                        37:  CALL   gt
                                                        38:  JUMPIF(1) 31
```



execute ⟦let
  var i:Integer
    in i := i+2⟧

elaborate ⟦var i:Integer⟧

execute ⟦i := i+2⟧

```
PUSH    1
LOAD    i
LOADL   2
CALL    add
STORE   i
POP(0)  1
```

$$\begin{array}{r} \mathit{execute}\ [\![\mathtt{let\ const\ n \sim 7;} \\ \mathtt{var\ i : Integer} \\ \mathtt{in\ i := n*n}]\!] \end{array} \left\{ \begin{array}{rl} \mathit{elaborate}\ [\![\mathtt{const\ n \sim 7}]\!] & \{\ \mathtt{LOADL}\ \ 7 \\ \mathit{elaborate}\ [\![\mathtt{var\ i : Integer}]\!] & \{\ \mathtt{PUSH}\ \ \ 1 \\ \mathit{execute}\ [\![\mathtt{i := n*n}]\!] & \left\{ \begin{array}{ll} \mathtt{LOAD} & n \\ \mathtt{LOAD} & n \\ \mathtt{CALL} & \mathit{mult} \\ \mathtt{STORE} & i \end{array}\right. \\ & \ \ \mathtt{POP(0)}\,2 \end{array}\right.$$

Exercício

Aplicar os code templates correspondentes e mostrar o código gerado para o trecho de programa:

```
if (a=0) then
       while (b<5) do
          begin
          a:=a*b;
          b:=b+1;
          end
else if (a>0) then
              begin
              b:=a+b+c;
              a:=g(a,b);
              end
       else f(a+b,a*b);
```

Exercício

Elaborar code templates para os comandos:

- `do <comando> while <expressão>`
- `for (<comando 1> ; <expressão> ; <comando 2>) <comando 3>`

Exemplificar a aplicação dos mesmos.

Exercício

Mostrar o código gerado para o programa abaixo, incluindo os endereços das variáveis:

```
programa p;
      var a : boolean;
      var b : integer;
      procedure q;
            var c,d : real;
            procedure r;
                  var e : integer;
                  begin
                  if e+d+b>0 then r else q;
                  end;
            begin
            r;
            end;
      begin
      q;
      end.
```