Eidgenössische
Technische Hochschule
Zürich

Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Federal Institute of Technology at Zurich

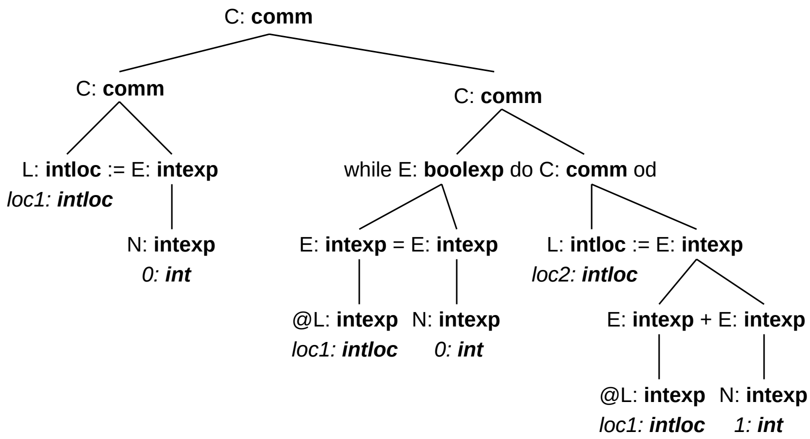
Michael Grossniklaus

Informatik 3

Winter Term 2001/2002

Week 1

Exercise 1.1:
The attributed syntax tree corresponding to the syntax tree of figure 2.2 in chapter 2 of the lecture script is shown below. The additionally included typing information is depicted in boldface.



Exercise 1.2:
To prove this exercise we define the following blackbox function $m(x)$. The function $m(x)$ will return $true$ iff $x \in C$ contains the same number of if as fi, and $false$ otherwise.

$$m(x) = \begin{cases} true & \#\text{if} = \#\text{fi} \\ false & \text{otherwise} \end{cases}$$

Now we have to check if $m(x) = true$ holds for all $x \in C$.

$L := E$	$m(x) = true$	as #if = #fi = 0
skip	$m(x) = true$	as #if = #fi = 0
if E **then** C_1 **else** C_2 **fi**	$m(x) = true$	as all of the above hold
while E **do** C **od**	$m(x) = true$	as all of the above hold
$C_1; C_2$	$m(x) = true$	as all of the above hold

As we can see, $m(x)$ returns $true$ in all possible cases. It is therefore not possible to write programs in $\mathcal{L}_c$ that contain a different number of if than fi.

Exercise 1.3:

(a) To evaluate $[[C]] = [[loc_1 := 1;\ loc_2 := @loc_1 + 1;\ \textbf{skip}]]$ we first note that C is of form $C_1; C_2; C_3$. Deriving from the script's semantic valuation function for $C_1; C_2$, we can find, that

$$[[C_1; C_2; C_3]] = [[C_3]]([[C_2]]([[C_1]](s)))$$

$s = \langle 0, 0, 0, 0\rangle$
$[[\textbf{skip} : comm]]([[loc_2 := @loc_1 + 1 : comm]]([[loc_1 := 1 : comm]](s)))$
$[[\textbf{skip} : comm]]([[loc_2 := @loc_1 + 1 : comm]](update([[loc_1 : intloc]], [[1 : intexp]](s), s)))$
$s = \langle 1, 0, 0, 0\rangle$
$[[\textbf{skip} : comm]]([[loc_2 := @loc_1 + 1 : comm]](s))$
$[[\textbf{skip} : comm]](update([[loc_2 : intloc]], [[@loc_1 + 1 : intexp]](s), s))$
$[[\textbf{skip} : comm]](update([[loc_2 : intloc]], plus([[@loc_1 : intexp]](s), [[1 : intexp]](s)), s))$
$[[\textbf{skip} : comm]](update([[loc_2 : intloc]], plus(lookup([[loc_1 : intloc]], s), [[1 : intexp]](s)), s))$
$[[\textbf{skip} : comm]](update([[loc_2 : intloc]], plus([[1 : intexp]](s), [[1 : intexp]](s)), s))$
$[[\textbf{skip} : comm]](update([[loc_2 : intloc]], [[2 : intexp]](s), s))$
$s = \langle 1, 2, 0, 0\rangle$
$[[\textbf{skip} : comm]](s)$
$s = \langle 1, 2, 0, 0\rangle$

(b) To evaluate $[[C]] = [[\textbf{while}\ @loc_1 = 0\ \textbf{do}\ loc_1 := 1\ \textbf{od}]]$ we follow the semantic valuation function given in the script.

$$[[\textbf{while}\ E\ \textbf{do}\ C\ \textbf{od}]] = w(s),\ \text{where}\ w(s) = if([[E : boolexp]](s), w([[C : comm]](s), s)$$

$s = \langle 0, 0, 0, 0\rangle$
$w(s) = if([[@loc_1 = 0 : boolexp]](s), w([[loc_1 := 1 : comm]](s)), s)$
$w(s) = if(equalint([[@loc_1 : intexp]](s), [[0 : intexp]](s)), w([[loc_1 := 1 : comm]](s)), s)$
$w(s) = if(equalint(lookup([[loc_1 : intloc]], s), [[0 : intexp]](s)), w([[loc_1 := 1 : comm]](s)), s)$
$w(s) = if(equalint([[0 : intexp]](s), [[0 : intexp]](s)), w([[loc_1 := 1 : comm]](s)), s)$
$w(s) = if(true, w([[loc_1 := 1 : comm]](s)), s)$
$w(s) = w([[loc_1 := 1 : comm]](s))$
$w(s) = w(update([[loc_1 : intloc]], [[1 : intexp]](s), s))$
$s = \langle 1, 0, 0, 0\rangle$
$w(s) = w(s)$
$w(s) = if([[@loc_1 = 0 : boolexp]](s), w([[loc_1 := 1 : comm]](s)), s)$
$w(s) = if(equalint([[@loc_1 : intexp]](s), [[0 : intexp]](s)), w([[loc_1 := 1 : comm]](s)), s)$
$w(s) = if(equalint(lookup([[loc_1 : intloc]], s), [[0 : intexp]](s)), w([[loc_1 := 1 : comm]](s)), s)$
$w(s) = if(equalint([[1 : intexp]](s), [[0 : intexp]](s)), w([[loc_1 := 1 : comm]](s)), s)$
$w(s) = if(false, w([[loc_1 := 1 : comm]](s)), s)$
$w(s) = s$
$s = \langle 1, 0, 0, 0\rangle$

Exercise 1.4:
To change $\mathcal{L}_c$ to handle boolean values, we have to extend the syntax rules, typing rules, semantic valuation functions and operations.

(i) Syntax Rules

$$B := true \mid false$$

$$E := N \mid ... \mid E_1 = E_2 \mid B$$

(ii) Typing Rules

$$\{true, false\} : bool$$

$$\frac{B : bool}{B : boolexp}$$

$$\frac{L : intloc}{@L : \tau exp}, \text{ for } \tau \in \{int, bool\}$$

$$\frac{L : intloc \qquad E : \tau exp}{L := E : comm}, \text{ for } \tau \in \{int, bool\}$$

(iii) Semantic Valuatuion Functions

$$[[B : boolexp]](s) = [[B : bool]]$$

$$[[B : bool]] = b$$

$$[[L := E : comm]](s) = update([[L : intloc]], [[E : \tau exp]](s), s), \text{ for } \tau \in \{int, bool\}$$

$$[[@L : \tau exp]](s) = lookup([[L : intloc]], s), \text{ for } \tau \in \{int, bool\}$$

(iv) Operations

$$Store = \{\langle n_1, n_2, \ldots n_m\rangle |\, n_i \in \{int, bool\}, 1 \leq i \leq m, m \geq 1\}$$

$$lookup : Location \times Store \rightarrow \{int, bool\}$$

$$update : Location \times \{int, bool\} \times Store \rightarrow Store$$

Exercise 1.5:

P Program $\rightarrow$ Nat*
$[[\text{ON } S]] = [[S]](0)$

Assumes initial memory cell value of 0.

S ExprSequence $\rightarrow$ Nat $\rightarrow$ Nat*
$[[E \text{ SUM } S]](n) = \text{let } n' = [[E]](n) \text{ in } n' \; cons \; [[S]](n')$
$[[E \text{ SUM OFF}]](n) = [[E]](n) \; cons \; nil$

It is important to have a definition that gives the value of an expression sequence as a list of integers, i.e. Nat $\rightarrow$ Nat*.

E Expression $\rightarrow$ Nat $\rightarrow$ Nat
$[[E_1 + E_2]](n) = plus([[E_1]](n), [[E_2]](n))$
$[[E_1 * E_2]](n) = times([[E_1]](n), [[E_2]](n))$
$[[E_1 - E_2]](n) = minus([[E_1]](n), [[E_2]](n))$

Assume *minus* defined over Nat such that if $n_1, n_2 \in$ Nat and $n_1 > n_2, n_2 - n_1 = 0$.

$[[\text{ IF } E_1, E_2, E_3]](n) = \text{if } [[E_1]](n) \text{ equals zero then } [[E_1]](n) \text{ else } [[E_2]](n)$
$[[\text{ IT }]](n) = n$
$[[(E)]](n) = [[E]](n)$
$[[N]](n) = [[N]]$

N Numeral $\rightarrow$ Nat
maps numeral N to corresponding $n \in$ Nat.

Exercise 1.6:
To prove this exercise we do a case analysis.

(i) $C = L := E$
The theorem is already proven for L (Propostion 2.2) and E (Proposition 2.3), therefore it holds here too.

(ii) $C =$ **skip**
The theorem is obviously true in this case.

(iii) $C = C_1; C_2$
The typing rule for a command sequence requires both C_1 and C_2 to be of type *comm*. It follows recursively that C is also of type *comm* and type *comm* is unique $\rightarrow$ (i), (ii).

(iv) $C =$ **while** E **do** C' **od**
If C is of type *comm* then the typing rule requires E to be of type *boolexp* and C' to be of type *comm*. It follows recursively that C is also of type *comm* and type *comm* is unique $\rightarrow$ (i), (ii).

(v) $C =$ **if** E **then** C_1 **else** C_2 **fi**
If C is of type *comm* then the typing rule requires E to be of type *boolexp* and C_1 and C_2 to be of type *comm*. It follows recursively that C is also of type *comm* and type *comm* is unique $\rightarrow$ (i), (ii).