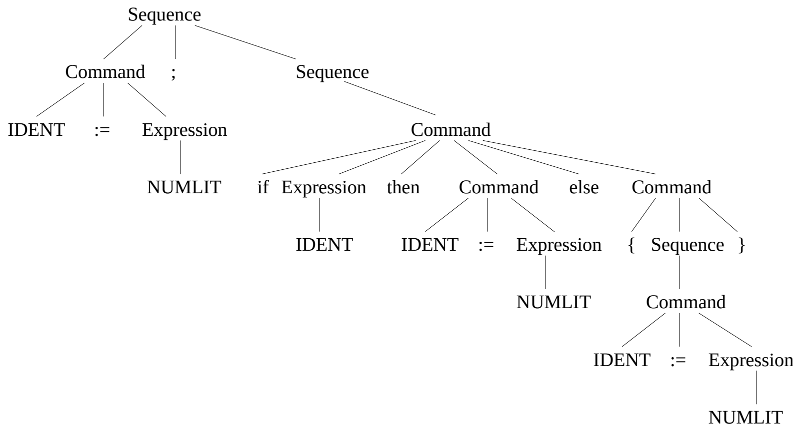
Programmation Avancée et Compilation 2001
(Solutions)

8 Visitors

Exercise 15

(a) In the same manner like in Exercise 13a, we construct the concrete syntax-tree:



(b) There is no unique way of defining the abstract syntax for a language. The abstract syntax should be simpler, and usually one would like to integrate those nonterminals into one that in the concrete grammar serve precedence purposes (such as Expression, Term, and Factor in Exercise 13). One possible abstract syntax for this exercise would be the following:

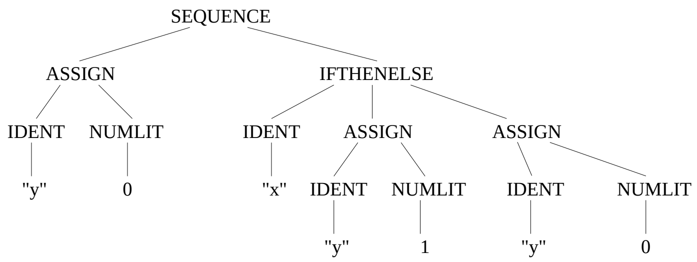
Sequence	::=	SEQUENCE { Sequence }
	|	IFTHENELSE Expression Sequence [Sequence]
	|	ASSIGN String Expression
Expression	::=	NUMLIT **int**
	|	BOOLLIT bool
	|	IDENT STRING

Another possible syntax would be the following:

Sequence	::=	SEMICOLON Sequence Sequence
	|	SHORTIF Expression Sequence
	|	LONGIF Expression Sequence Sequence
	|	ASSIGN String Expression
Expression	::=	...

Other alternatives could consist of a list of Sequence like in the first case but SHORTIF and LONGIF like in the second, or other combinations. Yet another alternative would be not even to distinguish between Sequence and Expression.

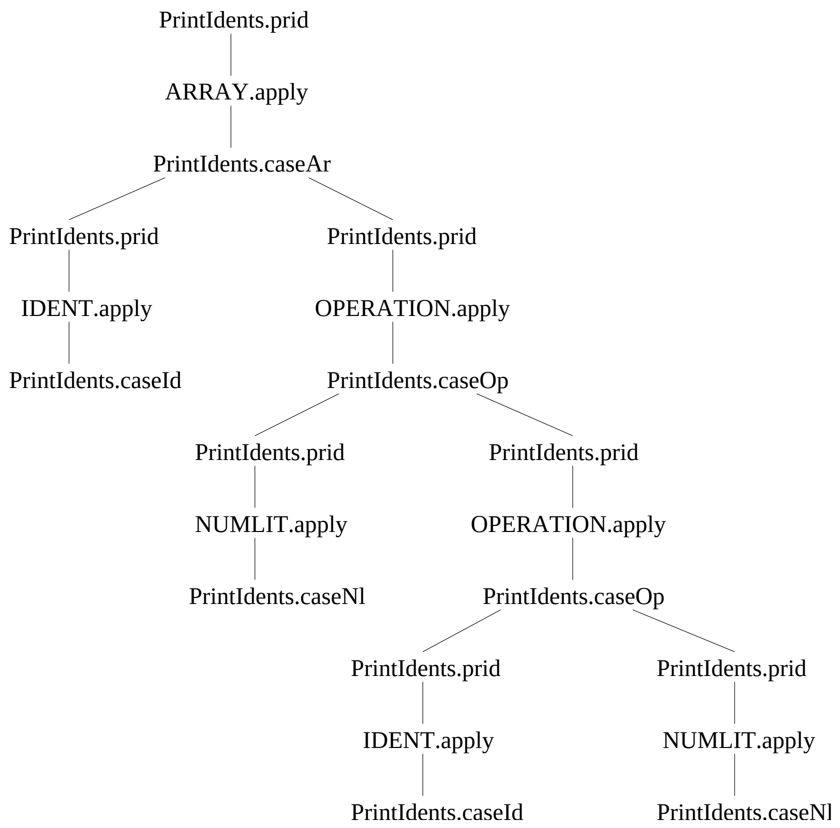
(c) We give the abstract syntax-tree according to our first grammar definition:



Exercise 16

(a) The grammar prints all identifiers occurring in a expression, one after the other. For the given example, this would yield `a i`.

(b) The example yields the following call-tree (we stick to the class terminology as presented in the lecture):



Note the (obvious) correspondence between the structure of the call-tree and that of the abstract syntax-tree. Recall from the lecture in which order the abstract syntax-tree is traversed while printing.