Semantics with Applications
Semantics of Expressions

Pablo López

University of Málaga

September 17, 2024

Outline

Outline

Abstract Syntax of WHILE (I)

We first define the syntactic categories of WHILE.

Abstract Syntax of WHILE (I)

We first define the syntactic categories of WHILE.

Syntactic categories and meta-variables

- $\mathbf{Num}$, n will range over numerals
- $\mathbf{Var}$, x will range over variables
- $\mathbf{Aexp}$, a will range over arithmetic expressions
- $\mathbf{Bexp}$, b will range over Boolean expressions, and
- $\mathbf{Stm}$, S will range over statements

Meta-variables can be primed (n') or subscripted (n_1).

Abstract Syntax of WHILE (II)

$$
\begin{array}{lcl}
a & ::= & n \mid x \mid a_1 + a_2 \mid a_1 \star a_2 \mid a_1 - a_2 \\
b & ::= & \texttt{true} \mid \texttt{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \wedge b_2 \\
S & ::= & x := a \mid \texttt{skip} \mid S_1 \texttt{ ; } S_2 \mid \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2 \\
 & \mid & \texttt{while } b \texttt{ do } S
\end{array}
$$

We assume:

- the usual (lexical) structure for numerals n and variables x
- the usual precedence for arithmetic operators
- liberal use of parentheses for grouping
- the usual textual representation for `&&`, `!`, `*`, `<=`

Abstract Syntax of WHILE (III)

Exercise. Identify the *basis* elements and *composite* elements for the syntactic categories $\mathbf{Aexp}$, $\mathbf{Bexp}$, and $\mathbf{Stm}$.

Exercise. Draw the parse tree for the program:

```
z := x;
x := y;
y := z
```

Abstract Syntax of WHILE (III)

Exercise. Identify the *basis* elements and *composite* elements for the syntactic categories $\mathbf{Aexp}$, $\mathbf{Bexp}$, and $\mathbf{Stm}$.

Exercise. Draw the parse tree for the program:

```
z := x;
x := y;
y := z
```

Did you miss something?

Abstract vs Concrete Syntax (I)

Abstract syntax is concerned with building syntactic constructions of the language with no ambiguity. It generates parse trees.
Concrete syntax is concerned with building **unique** parse trees out of textual representations of sentences.

Abstract vs Concrete Syntax (II)

Linear representation: `z:=x; (x:=y; y:=z)` vs `(z:=x; x:=y); y:=z`

Parse tree representation:

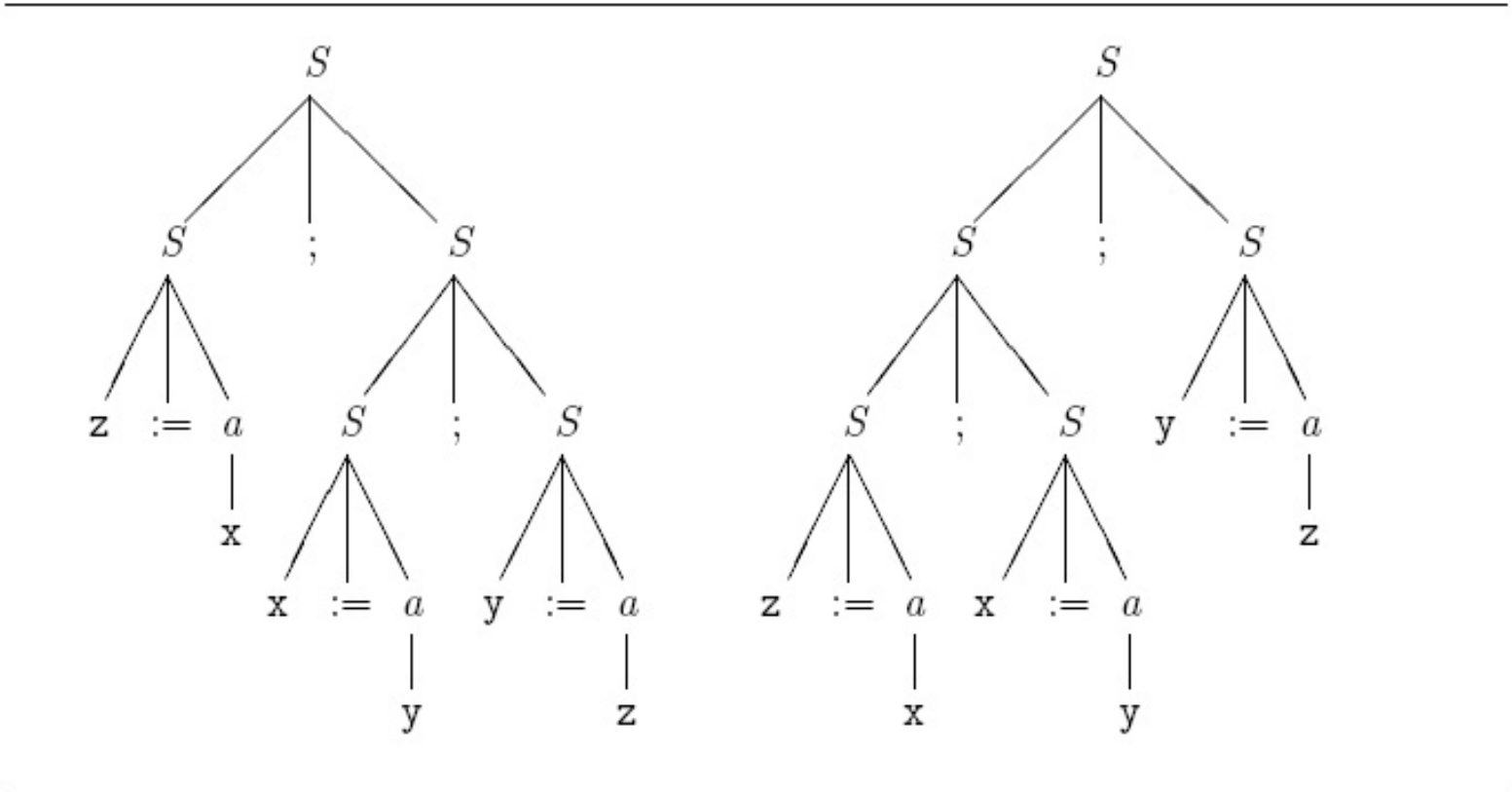

Figure 1.1: Abstract syntax trees for z:=x; x:=y; y:=z

Exercises

Exercise 1.1 The following statement is in WHILE:

```
y := 1;
while !(x=1) do (
    y := y*x;
    x := x-1
)
```

Draw the corresponding abstract syntax tree.

Exercise 1.2 Assume that the initial value of the variable x is n and that the initial value of y is m. Write a WHILE program that assigns to z the value of n^m.

Give a *linear* and a *graphical* representation of the abstract syntax.

Exercise. Define the appropriate types to represent the abstract syntax of arithmetic expressions $\mathbf{Aexp}$ in both Haskell and Java.

Exercise: From Abstract Syntax to Haskell

$$
\begin{array}{lcl}
a & ::= & n \mid x \mid a_1 + a_2 \mid a_1 \star a_2 \mid a_1 - a_2 \\
b & ::= & \texttt{true} \mid \texttt{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \wedge b_2 \\
S & ::= & x := a \mid \texttt{skip} \mid S_1 \,;\, S_2 \mid \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2 \\
 & \mid & \texttt{while } b \texttt{ do } S
\end{array}
$$

Represent the abstract syntax of $\mathbf{Aexp}$ in Haskell.

Exercise: From Abstract Syntax to Java

$$
\begin{array}{lcl}
a & ::= & n \mid x \mid a_1 + a_2 \mid a_1 \star a_2 \mid a_1 - a_2 \\
b & ::= & \texttt{true} \mid \texttt{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \wedge b_2 \\
S & ::= & x := a \mid \texttt{skip} \mid S_1 \texttt{ ; } S_2 \mid \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2 \\
 & \mid & \texttt{while } b \texttt{ do } S
\end{array}
$$

Represent the abstract syntax of **Aexp** in Java.

Outline

Semantic Functions

- A **semantic function** takes a syntactic entity as argument and returns its meaning.
- The semantics of `WHILE` is given by defining semantic functions for each of the syntactic categories.
- For $\mathbf{Num}$, $\mathbf{Aexp}$, and $\mathbf{Bexp}$, the semantic functions are specified once and for all.
- For $\mathbf{Var}$, we shall give two semantic functions.
- For $\mathbf{Stm}$, we could give separate operational, denotational, and axiomatic semantic functions. We shall focus on the operational semantics.

Example: Semantics for Binary Numerals (I)

Given the simplified syntactic category $\mathbf{Num}$:

```
n ::= 0 | 1 | n 0 | n 1
```

We define the semantic function:

$$\mathcal{N} : \mathbf{Num} \to \mathbf{Z}$$

$\mathcal{N}$ is a **total function**; for each numeral of $\mathbf{Num}$ returns a unique element in the semantic domain $\mathbf{Z}$; e.g.:

$$\mathcal{N}[\![101]\!] = \mathbf{5}$$

where $[\![$ and $]\!]$ are the *syntactic* brackets.

Example: Semantics for Binary Numerals (I)

Given the simplified syntactic category $\mathbf{Num}$:

```
n ::= 0 | 1 | n 0 | n 1
```

We define the semantic function:

$$\mathcal{N} : \mathbf{Num} \to \mathbf{Z}$$

$\mathcal{N}$ is a **total function**; for each numeral of $\mathbf{Num}$ returns a unique element in the semantic domain $\mathbf{Z}$; e.g.:

$$\mathcal{N}[\![101]\!] = \mathbf{5}$$

where $[\![$ and $]\!]$ are the *syntactic* brackets.
How do you define $\mathcal{N}$?

Example: Semantics for Binary Numerals (II)

$\mathcal{N} : \mathbf{Num} \to \mathbf{Z}$ is defined by the **semantic clauses** or **equations**:

$$\begin{aligned}
\mathcal{N}[\![\mathtt{0}]\!] &= \mathbf{0} \\
\mathcal{N}[\![\mathtt{1}]\!] &= \mathbf{1} \\
\mathcal{N}[\![n\ \mathtt{0}]\!] &= \mathbf{2} \cdot \mathcal{N}[\![n]\!] \\
\mathcal{N}[\![n\ \mathtt{1}]\!] &= \mathbf{2} \cdot \mathcal{N}[\![n]\!] + \mathbf{1}
\end{aligned}$$

- Beware the distinction between the syntactic domain ($\mathtt{0}$, $\mathtt{1}$) and the semantic domain ($\mathbf{0}$, $\mathbf{1}$).
- The definition of $\mathcal{N}$ is **syntax-directed** and **compositional**: for each possible way of constructing a numeral it tells how the corresponding number is obtained from the meanings of the **subconstructs**.

Exercises (I)

Exercise. Show a step by step evaluation of $\mathcal{N}[\![1010]\!]$.
Exercise. Argue whether compositional is equivalent to recursive.
Exercise. Implement the abstract syntax of binary numerals $\mathbf{Num}$ and the semantic function $\mathcal{N}$ in both Haskell and Java.

Exercise: From Semantic Functions to Haskell

$$
\begin{aligned}
\mathcal{N}[\![\mathtt{0}]\!] &= \mathbf{0} \\
\mathcal{N}[\![\mathtt{1}]\!] &= \mathbf{1} \\
\mathcal{N}[\![n\ \mathtt{0}]\!] &= \mathbf{2} \cdot \mathcal{N}[\![n]\!] \\
\mathcal{N}[\![n\ \mathtt{1}]\!] &= \mathbf{2} \cdot \mathcal{N}[\![n]\!] + \mathbf{1}
\end{aligned}
$$

Define the semantic function $\mathcal{N}$ in Haskell.

Exercise: From Semantic Functions to Java

$$
\begin{aligned}
\mathcal{N}[\![\texttt{0}]\!] &= \mathbf{0} \\
\mathcal{N}[\![\texttt{1}]\!] &= \mathbf{1} \\
\mathcal{N}[\![n\ \texttt{0}]\!] &= \mathbf{2} \cdot \mathcal{N}[\![n]\!] \\
\mathcal{N}[\![n\ \texttt{1}]\!] &= \mathbf{2} \cdot \mathcal{N}[\![n]\!] + \mathbf{1}
\end{aligned}
$$

Define the semantic function $\mathcal{N}$ in Java.

Exercises (II)

Exercise 1.4 Suppose that the grammar for n had been:

```
n ::= 0 | 1 | 0 n | 1 n
```

Define the corresponding semantic function $\mathcal{N}$. Implement the semantic function in both Haskell and Java.

Review of Mathematical Induction (I)

The set of natural numbers $\mathbb{N}$ is **inductively** defined by:

1. $0 \in \mathbb{N}$
2. if $n \in \mathbb{N}$ then $n + 1 \in \mathbb{N}$

Thus:

$$\mathbb{N} = 0, 0+1, 0+1+1, 0+1+1+1, \ldots$$

More commonly written as:

$$\mathbb{N} = 0, 1, 2, 3, \ldots$$

More formally, $\mathbb{N}$ is the *smallest* set that contains 0 and is closed under the *successor* operator.

Review of Mathematical Induction (II)

The *Principle of Mathematical Induction* states that to prove that a property P holds for every natural number:

$$\forall n \in \mathbb{N}.P(n)$$

it is sufficient to prove:

1. *Base case:* $P(0)$
2. *Induction step:* $\forall n \in \mathbb{N}.(P(n) \Rightarrow P(n+1))$

The assumption $P(n)$ in the induction step is called **Induction Hypothesis**.

Review of Mathematical Induction (II)

Use mathematical induction to prove the following statements:

1. $\sum_{i=1}^{n}(2i-1) = n^2$
2. $\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$
3. $\sum_{i=1}^{n} 2^{i-1} = 2^n - 1$

But first, take a look at our notation conventions. . .

Proof Strategy: Proving an Equality

To prove $A = B$, build a sequence of expressions that starts from A and finishes in B. Carefully justify the *reasoning* behind each step.

$$
\begin{aligned}
& A \\
= \quad & \{\textit{reasoning} \ldots\} \\
& A_1 \\
= \quad & \{\textit{reasoning} \ldots\} \\
& \vdots \\
= \quad & \{\textit{reasoning} \ldots\} \\
& A_n \\
= \quad & \{\textit{reasoning} \ldots\} \\
& B
\end{aligned}
$$

This notation is standard in PL and Haskell. Your proofs **must adhere** to this notation.

Why do we need Induction?

1. We shall use inductively defined sets
2. We shall prove properties on those sets
3. We shall prove those properties applying the appropriate induction principle (structural induction, induction on the length, rule induction, etc.)

The Set of Binary Numerals is Inductively Defined

The set of binary numerals $\mathbf{Num}$

```
n ::= 0 | 1 | n 0 | n 1
```

is **inductively** defined by:

1. $0 \in \mathbf{Num}$
2. $1 \in \mathbf{Num}$
3. if $n \in \mathbf{Num}$ then $n\ 0 \in \mathbf{Num}$
4. if $n \in \mathbf{Num}$ then $n\ 1 \in \mathbf{Num}$

$\mathbf{Num}$ is the *smallest* set that contains 0 and 1 and is closed under the operations that append a 0 or a 1.

Structural Induction on $\mathbf{Num}$

The *Principle of Structural Induction on* $\mathbf{Num}$ states that to prove that a property P holds for every binary numeral:

$$\forall n \in \mathbf{Num}.P(n)$$

it is sufficient to prove:

1. *Base Case:* $P(0)$
2. *Base Case:* $P(1)$
3. *Induction step:* $\forall n \in \mathbf{Num}.(P(n) \Rightarrow P(n\ 0))$
4. *Induction step:* $\forall n \in \mathbf{Num}.(P(n) \Rightarrow P(n\ 1))$

The assumption $P(n)$ in the induction steps is called **Induction Hypothesis**.

A Formal Result on $\mathcal{N}$

Theorem: The equations for $\mathcal{N}$ define a **total function**

$$\mathcal{N} : \mathbf{Num} \to \mathbf{Z}$$

Proof: We have a total function $\mathcal{N}$, if $\forall n \in \mathbf{Num}$ there is exactly one number $\mathbf{n} \in \mathbf{Z}$ such that $\mathcal{N}[\![n]\!] = \mathbf{n}$.
We proceed by **induction on the structure** of $\mathbf{Num}$.

A Useful Technique for Syntactic and Semantic Definitions

Compositional Definitions:

1. The syntax is specified by an **abstract syntax** giving *basis* and *composite* elements. The *composite* elements have a *unique* decomposition into their immediate constituents.
2. The semantics is defined by *compositional* definitions of a **semantic function**: There is a *semantic clause* for each of the *basis* elements, and one clause for each of the methods for constructing *composite* elements. The clauses for composite elements are defined in terms of the semantics of the immediate constituents.

A Powerful Technique for Proving Properties

Structural Induction:

1. Prove that the property holds for all the *basis* elements of the abstract syntax.
2. Prove that the property holds for all the *composite* elements of the abstract syntax: Assume that the property holds for all the immediate constituents of the element (this is the *induction hypothesis*) and prove that it also holds for the element itself.

Exercises (I)

Exercise. Define a syntactic category L to represent possibly empty sequences of arbitrary elements $x \in X$. Define an operator ++ that returns the concatenation of two such sequences. Prove whether your concatenation operator is associative.

Exercise. Define a syntactic category N to represent natural numbers. Define a semantic function $\mathbb{E}$ that determines whether a given natural number is even. State the *Principle of Structural Induction* for the syntactic category N. Prove that if a natural number n is even (odd), then its successor is odd (even).

Exercises (II)

Exercise. Define a syntactic category S to represent strings of arbitrary length of properly nested parentheses (e.g. `()`, `(())`, `(() ())`, `((())()((()))`). Define a semantic function $\mathbb{P}$ that returns a tuple with the number of left and right parentheses in a string $s \in S$. State the *Principle of Structural Induction* for the syntactic category S. Prove that the number of left parentheses is equal to the number of right parentheses.

From Structural Induction to Proving Properties of Haskell

Structural induction can be effectively applied to prove properties of Haskell programs.
Given:

```
(++) :: [a] -> [a] -> [a]
[] ++ ys = ys
(x:xs) ++ ys = x : xs ++ ys
```

prove that `(++)` is associative; i.e.:

$$\forall\ xs,\ ys,\ zs \in [\mathsf{a}].\ (xs +\!\!+ ys) +\!\!+ zs == xs +\!\!+ (ys +\!\!+ zs)$$

Proof: By induction on the structure of the lists. First state the principle of structural induction for lists in Haskell. Your proof **must adhere** to our notation for proving equalities.

Exercise

Given the standard definitions of `length`, `map` and `(.)`, prove the following statements:

$$\forall\ xs \in [\mathtt{a}],\ f \in \mathtt{a} \to \mathtt{b}.$$

$$\mathtt{length}\ xs == \mathtt{length}\ (\mathtt{map}\ f\ xs)$$

$$\forall\ xs \in [\mathtt{a}],\ f \in \mathtt{b} \to \mathtt{c},\ g \in \mathtt{a} \to \mathtt{b}.$$

$$(\mathtt{map}\ f\ .\ \mathtt{map}\ g)\ xs == \mathtt{map}\ (f\ .\ g)\ xs$$

The Concept of State

What is the value of `101`?

The Concept of State

What is the value of $\mathtt{101}$?

$$\mathcal{N}[\![\mathtt{101}]\!] = \mathbf{5}$$

The Concept of State

What is the value of `101`?

$$\mathcal{N}[\![101]\!] = \mathbf{5}$$

What is the value of `x + 1`?

The Concept of State

What is the value of `101`?

$$\mathcal{N}[\![101]\!] = \mathbf{5}$$

What is the value of `x + 1`?
Depends on the value of `x`...

The Concept of State

What is the value of `101`?

$$\mathcal{N}[\![101]\!] = \mathbf{5}$$

What is the value of `x + 1`?
Depends on the value of `x`...
The meaning of an expression depends on the **values** *bound to* the **variables** that occur in it.

The Concept of State

What is the value of `101`?

$$\mathcal{N}[\![101]\!] = \mathbf{5}$$

What is the value of `x + 1`?
Depends on the value of `x`...
The meaning of an expression depends on the **values** *bound to* the **variables** that occur in it.
The **state** associates to each variable its *current* value; i.e. it is a collection of bindings.

The Representation of the State

- As a function s of type $\mathbf{State} = \mathbf{Var} \to \mathbf{Z}$:

$$s\ \mathrm{x} = 5 \quad s\ \mathrm{y} = 7 \quad s\ \mathbf{z} = 0$$

The Representation of the State

- As a function s of type $\mathbf{State} = \mathbf{Var} \to \mathbf{Z}$:

$$s\ \mathtt{x} = 5 \quad s\ \mathtt{y} = 7 \quad s\ \mathtt{z} = 0$$

- As a table:

Variable	Value
x	**5**
y	**7**
z	**0**

The Representation of the State

- As a function s of type $\mathbf{State} = \mathbf{Var} \to \mathbf{Z}$:

$$s\ \mathrm{x} = 5 \quad s\ \mathrm{y} = 7 \quad s\ \mathrm{z} = 0$$

- As a table:

Variable	Value
x	$\mathbf{5}$
y	$\mathbf{7}$
z	$\mathbf{0}$

- As a list: $[\mathrm{x} \mapsto \mathbf{5}, \mathrm{y} \mapsto \mathbf{7}, \mathrm{z} \mapsto \mathbf{0}]$

Quiz

What is the most adequate representation for states?

1. Functions
2. Tables
3. Lists

Why?

Semantics of Arithmetic Expressions (I)

- From now on we assume the existence of a total function $\mathcal{N}$ defining the meaning of numerals in decimal base.
- We further assume that $s \in \mathbf{State}$ is a total function.
- Given an arithmetic expression $a \in \mathbf{Aexp}$ and a state $s \in \mathbf{State}$, we can determine the value $\mathbf{z} \in \mathbf{Z}$ of the expression a.
- The meaning of arithmetic expressions is defined by a total function $\mathcal{A}$:

$$\mathcal{A} : \mathbf{Aexp} \to (\mathbf{State} \to \mathbf{Z})$$

Note that $\mathcal{A}$ is *curried* (i.e. it takes parameters one at a time) and therefore higher-order.

Semantics of Arithmetic Expressions (II)

Recall that $\mathbf{Aexp}$ is defined as:

$$a ::= n \mid x \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 \star a_2$$

Then $\mathcal{A} : \mathbf{Aexp} \to \mathbf{State} \to \mathbf{Z}$ is defined by:

$$\begin{array}{lcl}
\mathcal{A}[\![n]\!]s & = & \mathcal{N}[\![n]\!] \\
\mathcal{A}[\![x]\!]s & = & s\ x \\
\mathcal{A}[\![a_1 + a_2]\!]s & = & \mathcal{A}[\![a_1]\!]s + \mathcal{A}[\![a_2]\!]s \\
\mathcal{A}[\![a_1 \star a_2]\!]s & = & \mathcal{A}[\![a_1]\!]s \cdot \mathcal{A}[\![a_2]\!]s \\
\mathcal{A}[\![a_1 - a_2]\!]s & = & \mathcal{A}[\![a_1]\!]s - \mathcal{A}[\![a_2]\!]s
\end{array}$$

Table 1.1: The semantics of arithmetic expressions

Beware:

- The distinction between the syntactic domain and the semantic domain
- The definition of $\mathcal{A}$ is **compositional**

Structural Induction on $\mathbf{AExp}$

Recall that $\mathbf{Aexp}$ is *inductively* defined as:

$$a ::= n \mid x \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 \star a_2$$

To prove a property P of arithmetic expressions:

$$\forall a \in \mathbf{Aexp}.\ P(a)$$

We can apply the *Principle of Structural Induction for* $\mathbf{Aexp}$:

1. Prove $\forall n \in \mathbf{Num}.\ P(n)$
2. Prove $\forall x \in \mathbf{Var}.\ P(x)$
3. Prove $\forall a_1, a_2 \in \mathbf{Aexp}.\ (P(a_1) \wedge P(a_2)) \Rightarrow P(a_1 + a_2)$
4. Prove $\forall a_1, a_2 \in \mathbf{Aexp}.\ (P(a_1) \wedge P(a_2)) \Rightarrow P(a_1 - a_2)$
5. Prove $\forall a_1, a_2 \in \mathbf{Aexp}.\ (P(a_1) \wedge P(a_2)) \Rightarrow P(a_1 \star a_2)$

Exercises

Exercise. What is the type of $\mathcal{A}[\![\mathtt{x}+1]\!]$?
Exercise. Suppose that $s\ \mathtt{x} = \mathbf{3}$. Evaluate stepwise $\mathcal{A}[\![\mathtt{x}+1]\!]s$.
Exercise 1.7 Prove that $\mathcal{A}$ is a total function. Hint: use structural induction.
Exercise. Suppose we add the arithmetic expression $-a$ to our language. Compare the two following semantic definitions:

$$\mathcal{A}[\![-a]\!]s = \mathbf{0} - \mathcal{A}[\![a]\!]s$$

$$\mathcal{A}[\![-a]\!]s = \mathcal{A}[\![0-a]\!]s$$

Are they equivalent?
Exercise. Add to the previous proof the case for $-a$ using both semantic definitions.
Exercise. Add the postfix increment operator ++ to $\mathbf{Aexp}$ and modify the definition of $\mathcal{A}$ accordingly.
Exercise. Implement $\mathcal{A}$ in both Java and Haskell.

Semantics of Boolean Expressions (I)

- The values of Boolean expressions are truth values in $\mathbf{T}$
- The semantic domain $\mathbf{T}$ consists of the truth values:
  - $\mathbf{tt}$ (for true), and
  - $\mathbf{ff}$ (for false)
- The meaning of Boolean expressions is defined by a total function $\mathcal{B}$:

$$\mathcal{B} : \mathbf{Bexp} \to (\mathbf{State} \to \mathbf{T})$$

Quiz

Given that $\mathcal{B}[\![b]\!]s$ receives a state s as a parameter:

$$\mathcal{B} : \mathbf{Bexp} \to (\mathbf{State} \to \mathbf{T})$$

Do we need to extend $\mathbf{State}$ to support Boolean variables and values?

Semantics of Boolean Expressions (II)

$\mathcal{B} : \mathbf{Bexp} \to \mathbf{State} \to \mathbf{T}$ is defined by:

$$
\begin{array}{lcl}
\mathcal{B}[\![\texttt{true}]\!]s & = & \mathbf{tt} \\
\mathcal{B}[\![\texttt{false}]\!]s & = & \mathbf{ff} \\
\mathcal{B}[\![a_1 = a_2]\!]s & = & \begin{cases} \mathbf{tt} & \text{if } \mathcal{A}[\![a_1]\!]s = \mathcal{A}[\![a_2]\!]s \\ \mathbf{ff} & \text{if } \mathcal{A}[\![a_1]\!]s \neq \mathcal{A}[\![a_2]\!]s \end{cases} \\
\mathcal{B}[\![a_1 \leq a_2]\!]s & = & \begin{cases} \mathbf{tt} & \text{if } \mathcal{A}[\![a_1]\!]s \leq \mathcal{A}[\![a_2]\!]s \\ \mathbf{ff} & \text{if } \mathcal{A}[\![a_1]\!]s > \mathcal{A}[\![a_2]\!]s \end{cases} \\
\mathcal{B}[\![\neg\ b]\!]s & = & \begin{cases} \mathbf{tt} & \text{if } \mathcal{B}[\![b]\!]s = \mathbf{ff} \\ \mathbf{ff} & \text{if } \mathcal{B}[\![b]\!]s = \mathbf{tt} \end{cases} \\
\mathcal{B}[\![b_1\ \wedge\ b_2]\!]s & = & \begin{cases} \mathbf{tt} & \text{if } \mathcal{B}[\![b_1]\!]s = \mathbf{tt} \text{ and } \mathcal{B}[\![b_2]\!]s = \mathbf{tt} \\ \mathbf{ff} & \text{if } \mathcal{B}[\![b_1]\!]s = \mathbf{ff} \text{ or } \mathcal{B}[\![b_2]\!]s = \mathbf{ff} \end{cases}
\end{array}
$$

Table 1.2: The semantics of boolean expressions

Once more, beware:

- The distinction between the syntactic and semantic domains
- The definition of $\mathcal{B}$ is **compositional**

Exercises

Exercise 1.8 Assume that $s\ \mathrm{x} = \mathbf{3}$, and determine $\mathcal{B}[\![\neg(\mathrm{x} = 1)]\!]$.

Exercise 1.9 State the Principle of Structural Induction for $\mathbf{Bexp}$ and prove that $\mathcal{B}$ is a total function.

Exercise. Define a syntactic category $\mathbf{Rexp}$ for relational expressions and define the corresponding semantic function $\mathcal{R}$. Argue whether compositional is equivalent to recursive, again.

Exercise 1.10 The syntactic category $\mathbf{Bexp}'$ is defined as the following extension of $\mathbf{Bexp}$:

$$\begin{array}{rcl} b & ::= & \mathtt{true} \mid \mathtt{false} \mid a_1 = a_2 \mid a_1 \neq a_2 \mid a_1 \leq a_2 \mid a_1 \geq a_2 \\ & \mid & a_1 < a_2 \mid a_1 > a_2 \mid \neg b \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \\ & \mid & b_1 \Rightarrow b_2 \mid b_1 \Leftrightarrow b_2 \end{array}$$

Give a *compositional* extension of the function $\mathcal{B}$ of Table 1.2. Two Boolean expressions b_1 and b_2 are *equivalent* if for all states s: $\mathcal{B}[\![b_1]\!]s = \mathcal{B}[\![b_2]\!]s$. Show that for each b' of $\mathbf{BExp}'$ there exists a Boolean expression b of $\mathbf{Bexp}$ such that b' and b are equivalent.

Outline

Free Variables

The **free variables** of an arithmetic expression a are defined to be the set of variables occurring in it. Formally, we may give a compositional definition of the subset $\mathrm{FV}(a)$ of $\mathbf{Var}$:

$$\begin{aligned}
\mathrm{FV}(n) &= \emptyset \\
\mathrm{FV}(x) &= \{\, x \,\} \\
\mathrm{FV}(a_1 + a_2) &= \mathrm{FV}(a_1) \cup \mathrm{FV}(a_2) \\
\mathrm{FV}(a_1 \star a_2) &= \mathrm{FV}(a_1) \cup \mathrm{FV}(a_2) \\
\mathrm{FV}(a_1 - a_2) &= \mathrm{FV}(a_1) \cup \mathrm{FV}(a_2)
\end{aligned}$$

Exercise. Compute the free variables of the following expressions:

- $x + 1$
- $x + y * x$
- $x \leq y$

What You Don't Know Won't Hurt You

Only the variables in $\mathrm{FV}(a)$ may influence the value of a.

Lemma 1.11 Let s and s' be two states satisfying that $s\ x = s'\ x$ for all x in $\mathrm{FV}(a)$. Then $\mathcal{A}[\![a]\!]s = \mathcal{A}[\![a]\!]s'$.

Proof. We proceed by structural induction.

Exercises

Exercise. Define the set of free variables $\mathrm{FV}(b)$ in a Boolean expression b.

Exercise 1.12 Let s and s' be two states satisfying that $s\ x = s'\ x$ for all x in $\mathrm{FV}(b)$. Prove that $\mathcal{B}[\![b]\!]s = \mathcal{B}[\![b]\!]s'$.

Substitutions (I)

A **substitution** $a[y \mapsto a_0]$ replaces each occurrence of a variable y in an arithmetic expression a with another arithmetic expression a_0.
Examples.

$$(\mathtt{x} + 1)[\mathtt{x} \mapsto 3] = 3 + 1$$

$$(\mathtt{x} * 2)[\mathtt{x} \mapsto 1 + 5] = (1 + 5) * 2$$

$$(\mathtt{x} + \mathtt{y} * \mathtt{x})[\mathtt{x} \mapsto \mathtt{y} - 5] = (\mathtt{y} - 5) + \mathtt{y} * (\mathtt{y} - 5)$$

Substitutions (II)

Substitutions are formally defined as follows:

$$
\begin{aligned}
n[y \mapsto a_0] &= n \\
x[y \mapsto a_0] &= \begin{cases} a_0 & \text{if } x = y \\ x & \text{if } x \neq y \end{cases} \\
(a_1 + a_2)[y \mapsto a_0] &= (a_1[y \mapsto a_0]) + (a_2[y \mapsto a_0]) \\
(a_1 \star a_2)[y \mapsto a_0] &= (a_1[y \mapsto a_0]) \star (a_2[y \mapsto a_0]) \\
(a_1 - a_2)[y \mapsto a_0] &= (a_1[y \mapsto a_0]) - (a_2[y \mapsto a_0])
\end{aligned}
$$

Exercise. Evaluate stepwise $((\mathtt{x} + 1) * (\mathtt{x} + \mathtt{y}))[\mathtt{x} \mapsto \mathtt{y} + 2]$.

Updates

An **update** $s[y \mapsto v]$ is a state that is like s except that the variable y is bound to the value v.

$$(s[y \mapsto\!\!\!\!\mid\; v])\ x = \begin{cases} v & \text{if } x = y \\ s\ x & \text{if } x \neq y \end{cases}$$

Exercises

Exercise 1.13 Prove that $\mathcal{A}[\![a[y \mapsto a_0]]\!]s = \mathcal{A}[\![a]\!](s[y \mapsto \mathcal{A}[\![a_0]\!]s])$ for all states s.

Exercise 1.14 Define substitution for Boolean expressions: $b[y \mapsto a_0]$ is to be the Boolean expression that is like b except that all occurrences of the variable y are replaced by the arithmetic expression a_0. Prove that your definition satisfies

$$\mathcal{B}[\![b[y \mapsto a_0]]\!]s = \mathcal{B}[\![b]\!](s[y \mapsto \mathcal{A}[\![a_0]\!]s])$$

for all states s.