Natural Semantics

Goals:

- Define the syntax of a simple imperative language
- Define a semantics using *natural deduction*[1]

[1]Natural deduction is an instance of first-order logic; that is, it is the formal language of first-order logic coupled with a "natural" deductive system based on proof trees.

A Simple Imperative Language

The following is the grammar $G = (\Gamma = T \cup N, \rightarrow, \gamma)$ for our simple imperative language with T and N as the obvious sets[2], $\gamma = C$ the start symbol (denoted as C^* in the rule set), and $\rightarrow$ defined as,

$$\begin{array}{rcl}
\mathrm{A} & \rightarrow & D \mid V \mid \mathrm{A} + \mathrm{A} \mid \mathrm{A} - \mathrm{A} \mid \mathrm{A} * \mathrm{A} \mid (\mathrm{A}) \\
\mathrm{B} & \rightarrow & \mathrm{T} \mid \mathrm{A} = \mathrm{A} \mid \mathrm{A} \leq \mathrm{A} \mid !\mathrm{B} \mid \mathrm{B}\&\&\mathrm{B} \mid \mathrm{B}||\mathrm{B} \mid (\mathrm{B}) \\
\mathrm{C}^* & \rightarrow & \textbf{skip} \mid \mathrm{V} := \mathrm{A} \mid \mathrm{C} \; ; \mathrm{C} \mid \textbf{if } \mathrm{B} \textbf{ then } \mathrm{C} \textbf{ else } \mathrm{C} \textbf{ end} \mid \textbf{while } \mathrm{B} \textbf{ do } \mathrm{C} \textbf{ end} \\
\mathrm{T} & \rightarrow & \textbf{true} \mid \textbf{false} \\
\mathrm{D} & \rightarrow & \mathrm{Q} \mid -\mathrm{Q} \\
\mathrm{Q} & \rightarrow & \mathbf{0}\,\mathrm{Q} \mid \ldots \mid \mathbf{9}\,\mathrm{Q} \mid \mathbf{0} \mid \ldots \mid \mathbf{9} \\
\mathrm{V} & \rightarrow & \mathbf{a}\,\mathrm{V} \mid \ldots \mid \mathbf{z}\,\mathrm{V} \mid \mathbf{a} \mid \ldots \mathbf{z}
\end{array}$$

[2] What are they?

Semantics

Our goal is to define a semantics for each syntactic component of our language so that composing the semantics of each component will then give us a semantics of programs written in that language.

Furthermore, we want to construct our models or semantics using predicates, relations, and functions; that means we need to describe our syntax using *sets*.[3]

We

- need a more mathematical view of syntax – *abstract syntax*
- ignore pragmatics like operator precedence and actual parsing
- introduce *syntactic sets* sometimes also called *syntactic domains* (not to be confused with semantic domains!)

[3] Read Sections 1.1 and 2.1 and 2.2 in the book by David Schmidt.

Semantics

As stated above, we want to construct our models using predicates, relations, and functions but when we look at our grammar G we only have a single set: $L(G)$,

$$L(G) = \{q \mid \gamma \Rightarrow^* q \wedge q \in T^*\}$$

There is not much we can do with this set because we don't see the individual syntactic structures in the strings on this set.

Idea: What if we introduce a set for each non-terminal - non-terminals tend to capture the structure of syntactic units.

Syntactic Sets

Grammar:

$$\begin{array}{rcl}
\mathrm{A} & \rightarrow & D \mid V \mid \mathrm{A} + \mathrm{A} \mid \mathrm{A} - \mathrm{A} \mid \mathrm{A} * \mathrm{A} \mid (\mathrm{A}) \\
\mathrm{B} & \rightarrow & \mathrm{T} \mid \mathrm{A} = \mathrm{A} \mid \mathrm{A} \leq \mathrm{A} \mid !\mathrm{B} \mid \mathrm{B}\&\&\mathrm{B} \mid \mathrm{B}||\mathrm{B} \mid (\mathrm{B}) \\
\mathrm{C}^* & \rightarrow & \textbf{skip} \mid \mathrm{V} := \mathrm{A} \mid \mathrm{C}\,;\mathrm{C} \mid \textbf{if}\ \mathrm{B}\ \textbf{then}\ \mathrm{C}\ \textbf{else}\ \mathrm{C}\ \textbf{end} \mid \textbf{while}\ \mathrm{B}\ \textbf{do}\ \mathrm{C}\ \textbf{end} \\
\mathrm{T} & \rightarrow & \textbf{true} \mid \textbf{false} \\
\mathrm{D} & \rightarrow & \mathrm{Q} \mid -\mathrm{Q} \\
\mathrm{Q} & \rightarrow & \mathbf{0}\,\mathrm{Q} \mid \ldots \mid \mathbf{9}\,\mathrm{Q} \mid \mathbf{0} \mid \ldots \mid \mathbf{9} \\
\mathrm{V} & \rightarrow & \mathbf{a}\,\mathrm{V} \mid \ldots \mid \mathbf{z}\,\mathrm{V} \mid \mathbf{a} \mid \ldots \mathbf{z}
\end{array}$$

Syntactic Sets:

- $\mathbf{T} = \{\textbf{true}, \textbf{false}\}$.
- $\mathbf{I} = \{q \mid \mathrm{D} \Rightarrow^* q \wedge q \in T^*\}$.
- $\mathbf{Loc} = \{q \mid \mathrm{V} \Rightarrow^* q \wedge q \in T^*\}$.
- $\mathbf{Aexp} = \{q \mid \mathrm{A} \Rightarrow^* q \wedge q \in T^*\}$.
- $\mathbf{Bexp} = \{q \mid \mathrm{B} \Rightarrow^* q \wedge q \in T^*\}$.
- $\mathbf{Com} = L(G) = \{q \mid \mathrm{C} \Rightarrow^* q \wedge q \in T^*\}$.

IMP - A Simple Imperative Language

Syntactic Sets (syntax only!!! not to be confused with the mathematical notions of integers, booleans, or variables, *etc.*):[4]

I	This set consists of all positive and negative integer **digits** including zero
T	Truth constants **true** and **false**.
Loc	Locations (variable names as strings).
Aexp	Arithmetic expressions
Bexp	Boolean expressions
Com	Commands

NOTE: $\mathbf{I} \neq \mathbb{I}$, where $\mathbb{I}$ is the set of all integer *values*.
NOTE: $\mathbf{T} \neq \mathbb{B}$, where $\mathbb{B}$ is the set of all boolean *values*.

[4] This material is based on the book by Glynn Winskel, "The Formal Semantics of Programming Languages."

Syntax meets Semantics

The two notes on the previous slide put us squarely in the middle of *syntax meets semantics*. The separation of these two is especially critical for numbers because our use of standard notation has blurred the fact that the number symbols we write are indeed only symbols that need some sort of interpretation. The fact that the notion of a number is independent of the symbols we write is highlighted by the fact that we can define the natural numbers without number symbols:

$$\begin{array}{l}\emptyset\\ \{\emptyset\}\\ \{\emptyset,\{\emptyset\}\}\\ \{\emptyset,\{\emptyset\},\{\emptyset,\{\emptyset\}\}\}\\ \{\emptyset,\{\emptyset\},\{\emptyset,\{\emptyset\}\},\{\emptyset,\{\emptyset\},\{\emptyset,\{\emptyset\}\}\}\}\\ \vdots\end{array}$$

Note that each set has precisely as many elements as the standard notation denotes. [5] The set of all these sets make up the natural numbers $\mathbb{N}$. Furthermore, the sets are ordered according to the subset relation,

$$\emptyset \subset \{\emptyset\} \subset \{\emptyset,\{\emptyset\}\} \subset \{\emptyset,\{\emptyset\},\{\emptyset,\{\emptyset\}\}\} \subset \cdots$$

[5] https://lutzhamel.github.io/CSC501/docs/number-systems-as-sets.pdf

Syntax meets Semantics

Since our standard notation is nothing but a syntactic representation we can write a grammar and then an interpretation for the symbols,

$$\begin{array}{lcl} \mathsf{N} & \to & \mathsf{D} \,|\, \mathsf{DN} \\ \mathsf{D} & \to & 1\,|\,2\,|\,3\,|\,4\,|\,5\,|\,6\,|\,7\,|\,8\,|\,9\,|\,0 \end{array}$$

with the *interpretation* over the set of all derived strings,

$$\begin{array}{lcl} 0 & \mapsto & \emptyset \\ 1 & \mapsto & \{\emptyset\} \\ 2 & \mapsto & \{\emptyset, \{\emptyset\}\} \\ 3 & \mapsto & \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\} \\ 4 & \mapsto & \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}, \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\}\} \\ \vdots & & \end{array}$$

Given this interpretation, what *value* does the symbol 22 represent.

Syntax meets Semantics

Why is this interesting? It shows that the mathematical notion of $\mathbb{N}$ is not tied to our syntactic representation. In fact, we could come up with a brand new way of writing the natural numbers by shifting the numeric keys on my computer keyboard:

$$\begin{array}{lcl} \mathsf{N} & \rightarrow & \mathsf{D} \mid \mathsf{DN} \\ \mathsf{D} & \rightarrow & ! \mid @ \mid \# \mid \$ \mid \% \mid \wedge \mid \& \mid * \mid (\mid) \end{array}$$

with the *interpretation*, call it *h*, over all derived strings,

$$\begin{array}{ccl}) & \mapsto & \emptyset \\ ! & \mapsto & \{\emptyset\} \\ @ & \mapsto & \{\emptyset, \{\emptyset\}\} \\ \# & \mapsto & \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\} \\ \$ & \mapsto & \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}, \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\}\} \\ \vdots & & \end{array}$$

Given this interpretation, what *value* does the symbol @@ represent.

Syntax meets Semantics

This idea can also be represented as the following diagram. Let G be a grammar for the syntax of some natural number symbols, then $L(G)$ is the set of all the symbols we can write, consider,

$$\begin{array}{rcl} \mathrm{N}^* & \rightarrow & \mathrm{D} \mid \mathrm{DN} \\ \mathrm{D} & \rightarrow & ! \mid @ \mid \# \mid \$ \mid \% \mid \wedge \mid \& \mid * \mid (\mid) \end{array}$$

Then an interpretation h as defined in the previous slide is

$$\begin{tikzcd} \mathbb{N} \\ L(G) \arrow[u, "h"] \end{tikzcd}$$

That is, the interpretation h takes each symbol in $L(G)$ and assigns it an element in $\mathbb{N}$ – assigns it *meaning*! We will see this as a recurring theme in this class.

Formation Rules for Syntactic Sets

Back to our small imperative language, we have the *deductively* defined syntactic sets: Syntactic Sets:

- $\mathbf{T} = \{\mathbf{true}, \mathbf{false}\}$.
- $\mathbf{I} = \{q \mid \mathrm{D} \Rightarrow^* q \wedge q \in T^*\}$.
- $\mathbf{Loc} = \{q \mid \mathrm{V} \Rightarrow^* q \wedge q \in T^*\}$.
- $\mathbf{Aexp} = \{q \mid \mathrm{A} \Rightarrow^* q \wedge q \in T^*\}$.
- $\mathbf{Bexp} = \{q \mid \mathrm{B} \Rightarrow^* q \wedge q \in T^*\}$.
- $\mathbf{Com} = L(G) = \{q \mid \mathrm{C} \Rightarrow^* q \wedge q \in T^*\}$.

There is another way to form the syntactic sets – *inductively* defined syntactic sets.

Formation Rules for Syntactic Sets

Let us consider the syntactic set **Aexp**. Our grammar production for arithmetic expressions is

$$\mathrm{A} \to D \,|\, V \,|\, \mathrm{A} + \mathrm{A} \,|\, \mathrm{A} - \mathrm{A} \,|\, \mathrm{A} * \mathrm{A} \,|\, (\mathrm{A})$$

We already know: $\mathbf{Aexp} = \{q \mid A \Rightarrow^* q \wedge q \in T^*\}$.

Given the above production we can define membership in the syntactic set as follows,

- $n \in \mathbf{Aexp}$ if $n \in \mathbf{I}$,
- $x \in \mathbf{Aexp}$ if $x \in \mathbf{Loc}$,
- $a_0 + a_1 \in \mathbf{Aexp}$ if $a_0, a_1 \in \mathbf{Aexp}$,
- $a_0 - a_1 \in \mathbf{Aexp}$ if $a_0, a_1 \in \mathbf{Aexp}$,
- $a_0 * a_1 \in \mathbf{Aexp}$ if $a_0, a_1 \in \mathbf{Aexp}$,
- $(a) \in \mathbf{Aexp}$ if $a \in \mathbf{Aexp}$.

This is an *inductive definition* of the set **Aexp**: we start with the "primitives" n and x and then build more and more complex expressions as part of **Aexp**.

It is easy to see that the inductive definition and the production rule describe the same structures, that is, given a production we can easily generate an inductive definition of the underlying syntactic set and, conversely, given an inductive definition of the syntactic set we can construct a production. *This means we can use productions and inductive set definitions interchangeably.*[6]

[6] Later on we will learn techniques that allow us to prove that these are equivalent definitions.

Formation Rules for Syntactic Sets

Observations for **Aexp**:

- $\mathbf{I} \subset \mathbf{Aexp}$,
- $\mathbf{Loc} \subset \mathbf{Aexp}$,
- contains structured *syntactic* objects that **look like** arithmetic expressions,

$$\mathbf{Aexp} = \{1, 5, x, \text{price}, 5 - 3 * y, x + 1, \ldots\}.$$

Formation Rules for Syntactic Sets

If the deductively defined set and the inductively defined set are equivalent, why bother introducing the inductively defined set?

Turns out many of our proofs have to establish properties over entire syntactic sets. The inductive nature of the syntactic sets allows us to use a proof technique called *structural induction* to show that a property holds for the entire syntactic set.

Evaluation of Arithmetic Expressions

Why are we so keen on representing syntax as sets?

We want to model computation denoted by the syntax in a mathematically meaningful way, that is, we want to use *functions* and *relations* to *interpret* the syntax and model a computation.

Functions and relations need sets as their domains and co-domains.

Evaluation of Arithmetic Expressions

In order to accomplish our goal we need define one additional entity. Let $\Sigma = \textbf{Loc} \rightarrow \mathbb{I}$ be the *set of all states*, where each element $\sigma \in \Sigma$ is a *state* and is considered a function of the form,

$$\sigma : \ \textbf{Loc} \rightarrow \mathbb{I}$$

where $\mathbb{I}$ represents the integer values. A state is a *function* such that $\sigma(x)$ will give the value of the *contents* for any $x \in \textbf{Loc}$.

The set Σ contains a distinguished state, σ_0, called the *initial state*, where $\sigma_0(x) = 0$, for all $x \in \textbf{Loc}$.

Evaluation of Arithmetic Expressions

We define the *evaluation* or *interpretation* for arithmetic expressions as a function from $\mathbf{Aexp} \times \Sigma$ into $\mathbb{I}$, that is, we define an evaluation call it $\mapsto$ as the function

$$\mapsto : \ \mathbf{Aexp} \times \Sigma \to \mathbb{I}$$

such that, given an arithmetic expression $a \in \mathbf{Aexp}$, some state $\sigma \in \Sigma$, and an appropriate value $k \in \mathbb{I}$, we write

$$(a, \sigma) \mapsto k$$

We say that the function evaluates an arithmetic expression a in the context of a state σ to the outcome k.

The pair (a, σ) is called a *configuration*.

Evaluation of Arithmetic Expressions

We need one more auxiliary function before we can define the actual evaluation function,

$$\mathit{eval} : \mathbf{I} \rightarrow \mathbb{I}$$

Given an element of the syntactic representation of numbers $\mathbf{I}$, this function will return the mathematical equivalent of this representation in $\mathbb{I}$.

Let $6 \in \mathbf{I}$, then $\mathit{eval}(6) = 6$, where the latter 6 is in $\mathbb{I}$. Another, perhaps more telling example is that our grammar allows us to generated numbers of the form ‘007’ for example. But $\mathit{eval}(007) = 7$. That is, $007 \in \mathbf{I}$ and $7 \in \mathbb{I}$.

We are not very formal about this and will drop this operation in later definitions, where it is clear from context that some $n \in \mathbf{I}$ represents a value in $\mathbb{I}$.

Natural Semantics

With all this machinery we are finally in the position to write down rules that define the evaluation function '$\mapsto$' for arithmetic expressions.

Observe that the rules mirror precisely the inductive definition of the set **Aexp**, in this way we are assured that the evaluation function is defined for all possible values/elements of the set **Aexp**.

The rules will be rules in the style of natural deduction.

Natural Deduction

Natural deduction is a formal system that has terms, functions, and relations as alphabet and that has two kinds of rules:

Inference Step

$$\frac{\text{premise}_1 \qquad \cdots \qquad \text{premise}_n}{\text{conclusion}}\ (\text{condition})$$

Assumption

$$\frac{}{\text{conclusion}}$$

NOTE: An inference step is valid if all its premises are true.
NOTE: An assumption is an inference step without premises (or with a premise that is always true).

Evaluation of Arithmetic Expressions

Evaluation of numbers

$$\frac{}{(n,\sigma) \mapsto eval(n)}$$

Evaluation of locations

$$\frac{}{(x,\sigma) \mapsto \sigma(x)}$$

Evaluation of sums

$$\frac{(a_0,\sigma) \mapsto k_0 \qquad (a_1,\sigma) \mapsto k_1}{(a_0 + a_1,\sigma) \mapsto k} \quad \text{where } k = k_0 + k_1$$

with k, k_0, $k_1 \in \mathbb{I}$, a_0, $a_1 \in$ **Aexp**, and $\sigma \in \Sigma$.

Evaluation of Arithmetic Expressions

Observation: We are abusing notation slightly. Technically, both the premises and the conclusions should evaluate to a boolean value which in our case is not true, the function $\mapsto$ is declared as,

$$\mapsto : \ \mathbf{Aexp} \times \Sigma \to \mathbb{I}$$

That is, it evaluates to an integer value. We can easily remedy this by turning the function into a *predicate*:

$$\mapsto : \ \mathbf{Aexp} \times \Sigma \times \mathbb{I} \to \mathbb{B}$$

so the rule for the evaluation of variable names would become something like,

$$\frac{}{(x, \sigma, \sigma(x)) \mapsto \mathbf{true}}$$

Which is a cumbersome notation and we will stick with our original notation remembering that we can always make this rigorous by turning the $\mapsto$ function into a predicate.

Evaluation of Arithmetic Expressions

Evaluation of subtractions

$$\frac{(a_0, \sigma) \mapsto k_0 \qquad (a_1, \sigma) \mapsto k_1}{(a_0 - a_1, \sigma) \mapsto k} \quad \text{where } k = k_0 - k_1$$

Evaluation of products

$$\frac{(a_0, \sigma) \mapsto k_0 \qquad (a_1, \sigma) \mapsto k_1}{(a_0 * a_1, \sigma) \mapsto k} \quad \text{where } k = k_0 \times k_1$$

Evaluation of parenthesized terms

$$\frac{(a, \sigma) \mapsto k}{((a), \sigma) \mapsto k}$$

with k, k_0, $k_1 \in \mathbb{I}$, a,a_0, $a_1 \in$ **Aexp**, and $\sigma \in \Sigma$.

Arithmetic Expression Summary

$$\frac{}{(n,\sigma) \mapsto eval(n)} \quad \text{for } n \in \mathbf{I}$$

$$\frac{}{(x,\sigma) \mapsto \sigma(x)} \quad \text{for } x \in \mathbf{Loc}$$

$$\frac{(a_0,\sigma) \mapsto k_0 \qquad (a_1,\sigma) \mapsto k_1}{(a_0 + a_1,\sigma) \mapsto k} \quad \text{where } k = k_0 + k_1$$

$$\frac{(a_0,\sigma) \mapsto k_0 \qquad (a_1,\sigma) \mapsto k_1}{(a_0 - a_1,\sigma) \mapsto k} \quad \text{where } k = k_0 - k_1$$

$$\frac{(a_0,\sigma) \mapsto k_0 \qquad (a_1,\sigma) \mapsto k_1}{(a_0 * a_1,\sigma) \mapsto k} \quad \text{where } k = k_0 \times k_1$$

$$\frac{(a,\sigma) \mapsto k}{((a),\sigma) \mapsto k}$$

with k, k_0, $k_1 \in \mathbb{I}$, a,a_0, $a_1 \in$ **Aexp**, and $\sigma \in \Sigma$.

Evaluation of Arithmetic Expressions

Observation: the line

with k, k_0, $k_1 \in \mathbb{I}$, a,a_0, $a_1 \in$ **Aexp**, *and* $\sigma \in \Sigma$.

on the previous slide and all our semantic definitions is absolutely necessary. We are dealing with first order logic and therefore all variables need to be quantified. Stating

$$k, k_0, k_1 \in \mathbb{I}$$

is the same thing as saying

$$\forall k, k_0, k_1 \in \mathbb{I}$$

That is, in all our semantic rules we have universal quantification. If we did not state this then the variables in the rules would be considered free variables and it possible to show that inferencing with free variables can get us into trouble.

Evaluation of Arithmetic Expressions

Observation: The inference rules are a structural definition of a function over the syntactic set **Aexp** which together with an appropriate state maps each syntactic element in that set into a value in the set of all integer values, $\mathbb{I}$, i.e., the meaning of an syntactic expression given some state is an integer value,

$$\begin{tikzcd}
\mathbb{I} \\
\textbf{Aexp} \times \Sigma \arrow[u]
\end{tikzcd}$$

Evaluation of Arithmetic Expressions

Example: Let $ae \equiv (2 * 3) + 5$, prove that the semantic value of this expression in some state σ is equal to 11 using the rules above

$$\dfrac{\dfrac{\dfrac{\dfrac{}{(2,\sigma) \mapsto \mathit{eval}(2) = 2} \qquad \dfrac{}{(3,\sigma) \mapsto \mathit{eval}(3) = 3}}{(2 * 3, \sigma) \mapsto 6}}{((2 * 3), \sigma) \mapsto 6} \qquad \dfrac{}{(5,\sigma) \mapsto \mathit{eval}(5) = 5}}{((2 * 3) + 5, \sigma) \mapsto 11}$$

$\Rightarrow$ The final result does not depend on the state – all constant expressions.

Evaluation of Arithmetic Expressions

Example: Now, let $ae \equiv v + 1$, where $v \in \mathbf{Loc}$, prove that the semantic value of this expression in some state $\sigma \in \Sigma$ is equal to $\sigma(v) + 1$.

$$\frac{\dfrac{}{(v, \sigma) \mapsto \sigma(v)} \qquad \dfrac{}{(1, \sigma) \mapsto 1}}{(v + 1, \sigma) \mapsto \sigma(v) + 1}$$

$\Rightarrow$ We cannot fully evaluate this expression because we don't know enough about the state σ.

Evaluation of Arithmetic Expressions

Example: Now, let $ae \equiv v + 1$, where $v \in \textbf{Loc}$, prove that the semantic value of this expression in the initial state $\sigma_0 \in \Sigma$ is equal to 1.

$$\frac{\dfrac{}{(v, \sigma_0) \mapsto \sigma_0(v) = 0} \qquad \dfrac{}{(1, \sigma_0) \mapsto 1}}{(v + 1, \sigma_0) \mapsto 1}$$

$\Rightarrow$ We can fully evaluate this expression because we know the definition of the initial state σ_0.