

Haskell Notation Reference

Every construct you need for Problem Set 1, and its semantic counterpart

CS 135: Computational Semantics | James Pustejovsky | Brandeis | Fall 2026

How to use this. Keep it open beside the scaffold. Section 1 is the one-page summary; Sections 2–5 explain each construct; Section 6 lists the mistakes that cost people the most time. [Nothing here is Haskell for its own sake](#) — every entry exists because it implements something from the semantics.

1. The Correspondence, at a Glance

Semantics	Haskell	Where it appears
type e	Entity	entities in the domain
type t	Bool, aliased T	sentence denotations
type $\langle e, t \rangle$	Entity $\rightarrow$ T, aliased ET	nouns, intransitive verbs
type $\langle e, \langle e, t \rangle \rangle$	Entity $\rightarrow$ Entity $\rightarrow$ T, aliased TV	transitive verbs
type $\langle \langle e, t \rangle, t \rangle$	ET $\rightarrow$ T, aliased GQ	quantified NPs
determiner type	ET $\rightarrow$ GQ, aliased DET	<i>every, some, no</i>
domain $\mathcal{D}$	domain :: [Entity]	what quantifiers range over
interpretation I	the ...Set lists and likePairs	the model
characteristic function	inSet, inRel	sets $\rightarrow$ functions
$\lambda x. \varphi$	\x -> ...	every denotation you write
function application	juxtaposition: f x	the composition rule
$\forall x$	all p domain	universal quantification
$\exists x$	any p domain	existential quantification
$P \rightarrow Q$	not p q	inside <i>every</i>
$P \wedge Q$	p && q	inside <i>some</i> , and PM
$\neg P$	not p	inside <i>no</i>

2. Declaring Things

Type signature :: Read :: as “has type”. Always write one; it is the semantic type of the expression, and it is what the compiler checks your definition against.

```
student :: ET      -- student has type <e,t>
likes   :: TV      -- likes has type <e,<e,t>>
```

New type data Creates a genuinely new type by listing its values. The values *are* the type — there is nothing underneath.

```
data Entity = Alice | Bob | Carol | Fido
```

deriving (Eq, Ord, Show, Enum, Bounded)

Read `|` as “or”. **Semantically this is the domain $\mathcal{D}$:** declaring the type declares what exists in the model.

deriving asks the compiler to generate boilerplate: `Eq` gives `==` (needed by `elem`), `Enum` and `Bounded` give `minBound`, `maxBound`, and range notation (needed by `domain`), `Show` gives printing.

Synonym type Creates a *nickname*, not a new type. `T` and `Bool` are interchangeable everywhere, and GHC may report errors using either name.

```
type T    = Bool           -- t
type ET   = Entity -> T    -- <e,t>
type TV   = Entity -> Entity -> T -- <e,<e,t>>
```

The synonyms exist so signatures read like semantics rather than like plumbing.

3. Functions

The arrow `->` `a -> b` is the type of functions from `a` to `b`.

It associates to the right. So

```
Entity -> Entity -> Bool    means    Entity -> (Entity -> Bool)
```

Give me an entity, and I hand back a one-place predicate. That is currying, visible in the type — and it is why partial application yields a verb phrase.

Lambda `\x -> ...` The backslash is λ . `\x -> e` is $\lambda x.e$.

```
sleeps = \x -> x 'elem' sleepSet    -- lambda style
sleeps x = x 'elem' sleepSet        -- equation style; identical
```

`\x y -> ...` abbreviates `\x -> \y -> ...` — two nested lambdas.

Application Juxtaposition. No parentheses, no commas.

It associates to the left: `f x y` means `(f x) y`.

```
sleeps Alice    -- one argument    : Bool
likes Bob       -- partial!         : Entity -> Bool
likes Bob Alice -- both arguments   : Bool
```

The middle line is the important one: applying a `TV` to one argument gives an `ET` — the type of a verb phrase. Partial application *is* phrase structure.

Point-free definitions A definition need not mention its argument, if the right-hand side is already a function:

```
student = inSet studentSet    -- point-free
student x = x 'elem' studentSet -- equivalent, argument named
```

Both define the same `ET`. The first makes visible that a predicate is an object you can build and pass around.

4. Data and Lookup

Lists [] [Entity] is the type of lists of entities; [Alice, Bob] is such a list.

```
domain :: [Entity]
domain = [minBound .. maxBound]    -- range notation: [Alice, Bob, Carol, Fido]
```

Writing `domain` this way rather than by hand means it stays correct when you add to the `data` declaration.

Tuples (,) (Entity, Entity) is an ordered pair, so [(Entity, Entity)] is a list of pairs — a binary relation.

```
likePairs = [(Alice,Bob), (Alice,Fido), (Bob,Fido)]
```

Order matters: (Alice,Bob) is present, (Bob,Alice) is not.

elem and backticks `elem` tests list membership. Backticks put a two-argument function between its arguments:

```
elem x xs      -- prefix
x `elem` xs    -- infix; identical, and reads like x an element of xs
```

inSet and inRel These are given in the scaffold. They convert the model from *sets* into *functions*.

```
inSet :: [Entity] -> ET
inSet xs = \x -> x `elem` xs

inRel :: [(Entity,Entity)] -> TV
inRel ps = \obj subj -> (subj, obj) `elem` ps
```

inSet is the characteristic function $\chi_S(x) = 1 \iff x \in S$ — the set/function isomorphism, in four words.

inRel takes the object first but stores pairs subject-first. It is the adapter between the two conventions. See the trap in §6.

5. Logic

Booleans

Logic	Haskell	Note
$P \wedge Q$	<code>p && q</code>	
$P \vee Q$	<code>p q</code>	
$\neg P$	<code>not p</code>	<code>not</code> is a function; parenthesize its argument
$P \rightarrow Q$	<code>not p q</code>	no arrow operator exists

```
all :: (a -> Bool) -> [a] -> Bool    -- every element passes
any :: (a -> Bool) -> [a] -> Bool    -- at least one passes

forall p = all p domain             -- the clause for forall x
exists p = any p domain              -- the clause for exists x
```

These are the satisfaction clauses of Session 3, transcribed. The assignment update $g[x \mapsto d]$ is Haskell’s function application `p d`: because `p` is a *function* of x , applying it to d is evaluating φ under the modified assignment.

This works because `domain` is finite. An infinite domain would make `all` diverge.

```
every :: DET
every restrictor scope = all (\x -> not (restrictor x) || scope x) domain

some' :: DET
some' restrictor scope = any (\x -> restrictor x && scope x) domain
```

Note the asymmetry, and do not swap it. *every* pairs with implication; *some* and *no* pair with conjunction. Swapping produces no type error and the wrong truth conditions — see §6.

Primes on `some'` and `no'` avoid clashing with Prelude names. Same for `read'`.

6. Traps

Determiners

1. Argument order on transitive verbs. `likes Bob Alice` means *Alice likes Bob*. Object first. This typechecks either way, so the compiler will not save you — you will simply get FAIL on the test.

Check: `likes obj subj` looks up `(subj, obj)` in the pair list.

2. Wrong connective in a determiner. `all (\x -> q x && p x) domain` for *every* claims that *everything in the domain* is a Q that is P — not that every Q is P . Typechecks; wrong.

Likewise implication inside *some* makes *some dog sleeps* true whenever anything fails to be a dog.

3. Missing parentheses after not. Application binds tightest, so `not student x` parses as `(not student) x` — applying `not` to a function. Write `not (student x)`.

4. Bypassing the composition rules. `s3 = likes bob alice` may give the right answer, but Part D asks for `spName alice (vpName likes bob)`. The rules are the point of the assignment; shortcuts receive partial credit.

5. Applying an entity to something. `alice sleeps` is a type error: `alice :: Entity` is not a function. The predicate applies to the individual, so write `sleeps alice`.

7. Reading a Type Error

GHC error messages have a fixed shape. Find these two lines and compare them:

```
* Couldn't match expected type: Entity
    with actual type: Entity -> Bool
* In the first argument of 'tall', namely 'student'
```

Expected is what the position demands. **Actual** is what you supplied. The gap between them names the failing composition step — here, an $\langle e, t \rangle$ where an e was required, which is exactly the mismatch that motivates predicate modification.

Message	Usually means
applied to one value argument, but its type has none	you applied a non-function, e.g. an <code>Entity</code>
expected <code>Entity</code> , actual <code>Entity -> Bool</code>	you passed a predicate where an individual was wanted
expected <code>Bool</code> , actual <code>Entity -> Bool</code>	you forgot to apply a predicate to anything
Variable not in scope	typo, or a Prelude clash (try a prime)

Read the first error only. Later ones are usually cascade damage from the first.

8. Working in GHCi

```
$ runghc Fragment.hs      -- run the test suite
$ ghci Fragment.hs        -- load interactively
```

Inside GHCi:

```
:t expr  show the type of an expression — the most useful command
:r        reload after editing the file
:q        quit
```

```
> :t likes Bob
likes Bob :: Entity -> Bool      -- confirms partial application gives an ET
> student Alice
True
> every student sleeps
False
```

Ask the compiler when you are stuck. Put a hole where you cannot see what belongs:

```
myDenotation = _
```

GHC will report the type it expects there, which is often enough to finish the line.

If a hole is taking you more than three or four lines, stop — in the reference solution every hole is a single line, and you are probably fighting the language rather than doing the semantics. Bring it to lab or office hours.