# Quiz 2 Study Guide

---

**Semantics**

- Given an abstract syntax and an informal description of the semantics, give an appropriate semantic domain (as a Haskell type).
- Implement a denotational semantics for a simple language in Haskell.

## Defining a language with denotational semantics

Example encoding in Haskell:

1. Define the **abstract syntax**, $T$ — `data Term = ...`
   *the set of abstract syntax trees*
2. Identify or define the **semantic domain**, $V$ — `type Value = ...`
   *the representation of semantic values*
3. Define the **valuation function**, $[\![\cdot]\!] : T \to V$ — `sem :: Term -> Value`
   *the mapping from ASTs to semantic values*

**EXAMPLE #1:**

**Consider the following language for implementing a simple counter. Statements either increment the counter by a given integer, or they reset the counter to zero. A program runs a sequence of statements on an initial counter of 0 and returns the final value of the counter.**

```
i ::= (any integer)
s ::= inc i
   |  reset
p ::= s ; p
   |  ε
```

**Abstract Syntax:**

```
type i = int
data s = inc i
       | reset
   deriving(Eq, Show)
```

**What is the Semantic Domain?**

**EXAMPLE #2:**

**Consider the following command language for controlling a robot that moves in a one-dimensional space (i.e. back and forth along a line).**

```
data Cmd = Gas | Brake | Turn
type Prog = [Cmd]
```

**The state of the robot is represented by three components: its current position on the line, its current direction, and its speed. Moving forward corresponds to increasing the position while moving backward decreases the position.**

**The commands work as follows:**

```
type Pos   = Int
type Speed = Int
data Dir   = Forward | Backward
```

- **Gas: Move in the current direction an amount equal to the current speed, then increase the speed by one. For example, if the robot is at position 5 while moving forward at a speed of 2, after executing a Gas command the robot would be at position 7 moving at a speed of 3.**
- **Brake: Move in the current direction an amount equal to the current speed, then decrease the speed by one down to a minimum speed of 0. If the robot is already at speed 0, then a Brake command has no effect.**
- **Turn: If the current speed of the robot is 0, then change the direction of the robot. If the speed is not 0, the robot crashes. If the robot crashes, it no longer has a speed or direction, but it does still have a position (the position it was at when it crashed).**

**Abstract Syntax**

```
type Pos      = Int
type Speed    = Int
data Dir      = Forward
              | Backward
     deriving (Eq, Show)
data Cmd      = Gas
              | Brake
              | Turn
     deriving (Eq, Show)
type prog = [Cmd]
```

**What is the Semantic Domain?**

**Types**

- Static vs. dynamic typing: what is the difference, what are the tradeoffs?

| **Static** | **Dynamic** |
|---|---|
| - External names refer to variables that are visible at **definition.**<br>- Only supports planned extensibility<br>- Names are not part of the public interface<br>    - No risk of name collision<br>    - Improved modularity | - External names refer to variables that are visible at **call site.**<br>- Supports ad-hoc extensibility<br>- All names are part of the public interface<br>    - Risk of name collision<br>    - Bad modularity |

- Implement a typing relation for a simple language in Haskell.

## Defining a static type system

Example encoding in Haskell:

1. Define the **abstract syntax**, $E$ — `data Exp = ...`
   *the set of abstract syntax trees*
2. Define the structure of **types**, $T$ — `data Type = ...`
   *another abstract syntax*
3. Define the **typing relation**, $E : T$ — `typeOf :: Exp -> Type`
   *the mapping from ASTs to types*

Then, we can define a dynamic semantics that **assumes** there are no type errors

**EXAMPLE #1: IntBool.hs**

**Abstract syntax**

```
data Exp
    = Lit Int
    | Add Exp Exp
    | Mul Exp Exp
    | Equ Exp Exp
    | If  Exp Exp Exp
  deriving (Eq, Show)
```

**Types**

```
data Type = TBool | TInt | TError
    deriving (Eq, Show)
```

**Define the Typing Relation.**

**EXAMPLE #2: Imp.hs**

**Abstract Syntax**

```
data Expr
        = Lit Int
        | Add Expr Expr
        | LTE Expr Expr
        | Not Expr
        | Ref Var
deriving (Eq, Show)

data Stmt
        = Bind Var Expr
        | If Expr Stmt Stmt
        | While Expr Stmt
        | Block [Stmt]
deriving (Eq, Show)
```

**Types**

```
data Type = TInt | TBool
        deriving (Eq, Show)
```

**Identify Type Expression**

**Identify Type Statement**

**Naming**

- In a snippet of Haskell code, identify all of the name declarations and all of the name references.

## What is naming?

Most languages provide a way to **name** and **reuse** stuff

### Naming concepts

| | |
|---|---|
| **declaration** | introduce a new name |
| **binding** | associate a name with a thing |
| **reference** | use the name to stand for the bound thing |

### C/Java variables

```
int x; int y;
x = slow(42);
y = x + x + x;
```

**In Haskell:**

### Local variables

```
let x = slow 42
in x + x + x
```

### Type names

```
type Radius = Float
data Shape = Circle Radius
```

### Function parameters

```
area r = pi * r * r
```