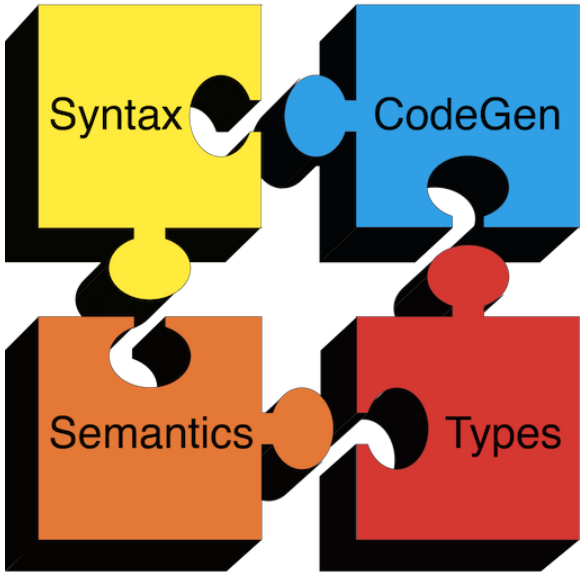


Compilers

Self Type

Self Type

```
class Count {
  i : int ← 0;
  inc () : Count {
       {
          i ← i + 1;
          self;
       }
    };
};
```

- Class Count incorporates a counter
- The inc method works for any subclass

Self Type

- Consider a subclass Stock of Count

```
class Stock inherits Count {
  name : String; -- name of item
};
```

inc(): Count

- And the following use of Stock:

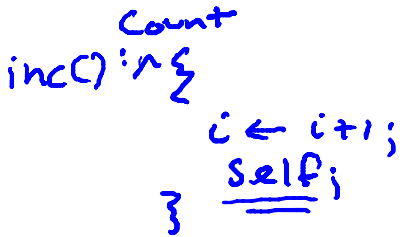
```
class Main {
  Stock a ← (new Stock).inc ();
  … a.name …
};
```

Count

Count ≠ Stock

Type error!

Self Type



- (new Stock).inc() has dynamic type Stock
- So it is legitimate to write

  Stock a ← (new Stock).inc ()
- But this is not well-typed
  - (new Stock).inc() has static type Count
- The type checker "loses" type information
  - This makes inheriting inc useless
  - So, we must redefine inc for each of the subclasses, with a specialized return type

Self Type

- We will extend the type system
- Insight:
  - inc returns “self”
  - Therefore the return value has same type as “self”
  - Which could be Count or any subtype of Count!
- Introduce the keyword SELF_TYPE to use for the return value of such functions
  - We will also need to modify the typing rules to handle SELF_TYPE

Self Type

- SELF_TYPE allows the return type of inc to change when inc is inherited

- Modify the declaration of inc to read

  inc() : SELF_TYPE { … }

- The type checker can now prove:

  O,M,C ⊢ (new Count).inc() : Count

  O,M,C ⊢ (new Stock).inc() : Stock

- The program from before is now well-typed

Self Type

is not a class name

- SELF_TYPE is not a dynamic type
  - It is a static type
  - It helps the type checker to keep better track of types
  - It enables the type checker to accept more correct programs
- In short, having SELF_TYPE increases the expressive power of the type system