Language Engineering - Reviewed Notes

Dom Hutchinson

April 12, 2019

Contents

Not Covered.
Below are some *advanced* topics that should only be looked at if happy with everything else

- There is an alternative proof technique (to the one discussed in **2.2**) in **Chapter 3.4** of the book.
- There are extensions to the *While* language which define *Procedures, Locations* & *Continuations.*

1 Syntax

1.1 Domain Specific Languages & Catamorphisms

1.1.1 Domain Specific Languages

This subsection was covered in Teaching Block 1; Lectures 1-4.

Definition 1.1 - *Domain Specific Language*
A *Domain Specific Language* is a programming language that has been design for a specific purpose.
Domain Specific Languages are not necessarily *Turing Complete*.
E.G. - *SQL* as an example of a *Domain Specific Language* for database management.
N.B. - The term *Domain Specific Languages* is often abbreviated to *DSL*.

Definition 1.2 - *Embedded Domain Specific Language*
An *Embedded Domain Specific Language* is a *Domain Specific Language* which has been defined within a host language.
There are two techniques for embedding a language within the host

i) Deep Embedding; and,

ii) Shallow Embedding.

E.G. - Packages within programming languages can be considered as *Embedded Domain Specific Languages*.
N.B. - The term *Embedded Domain Specific Languages* is often abbreviated to *EDSL*.

Remark 1.1 - *DSL v EDSL*
When defining a language as an *EDSL* you are restricted by the language you are using, this is not the case for a *DSL* definition.
However, when creating a *DSL* definition you have to produce your own parsers & compilers which consume a lot of time.
Definition 1.3 - *Deep Embedding*
In a *Deep Embedding* of an *EDSL* we define the syntax with concrete data types & the semantics with functions that evaluate these data types.

Definition 1.4 - *Concrete Data Type*
A *Concrete Data Type* is absolutely defined with only certain inputs & outputs allowed.
E.G. - Boolean, Integer, Arrays & Lists.

Example 1.1 - *Concrete Data Types*
Consider a syntax for adding multiple integers together *e.g.* $3 + 5$.
We have two main features, the values & the addition symbol, thus we need to define expressions within the constructor.
We know the values are strictly integers & the addition symbol only operates on two values, thus we define

```
data Expr = Var Int
          | Add Expr Expr
```

Then $1 + (4 + 7) \equiv$Add (Var 1) (Add (Var 4) (Var 7)).

Definition 1.5 - *Deep Embedding Semantics*
In a *Deep Embedding* the *Semantics* are defined as functions which evaluate the defined data

types.
This means they take in a single value of the data type & produce a single value.

Example 1.2 - *Deep Embedding Semantics*
Consider define a semantics for the situation describe in **Example 1.1**.
We want to produce an Int value from Expr.
We need to define what the evaluation function does for each construction of Expr.
We break down the Add constructions & analyse their sub-structures.

```
eval :: Expr -> Int
eval (Var n)   = n
eval (Add x y) = eval x + eval y
```

We then analyse expressions by passing them as inputs to eval.
Definition 1.6 - *Shallow Embedding*
In a *Shallow Embedding* of an *EDSL* we use the syntax of the host language to define functions that produce results directly.
Data types in a *Shallow Embedding* are represented by functions.
N.B. - There is no need for evaluating functions in a *Shallow Embedding*.

Example 1.3 - *Shallow Embedding*
Consider define a *Shallow Embedding* for an EDSL that adds integers together.[1]
We have two main features, the values & the addition symbol, thus we need to define functions for each of them.
We shall define them to always evaluate to Ints for this example

```
var :: Int -> Int
var n = n
add :: Int -> Int -> Int
add x y = x + y
```

Then $1 + (4 + 7) \equiv$add (var 1) (add (var 3) (var 5)).

1.1.2 Fix & Catamorphisms

This subsection was covered in Teaching Block 1; Lecture 4-6.

Remark 1.2 - *Expression Problem*
The *Expression Problem* concerns whether it is possible to extend the syntax & semantics of a language in a modular way.
i.e. Can we define a language that is easy to extend both the syntax & semantics of.
Hard & Soft Embedding allow for one of these, but not both.
The *Expression Problem* is solved using *Catamorphisms*
Example 1.4 - *`foldr`* *- List*
Below is a definition for `foldr` to apply a function to a list

```
foldr :: b -> (a->b->b) -> [a] -> b
foldr k f []     = k
foldr k f (x:xs) = f x (foldr k xs)
```

Definition 1.7 - *`Functor`*
`Functors` are a class of data types that allow you to work inside the data-structure.
`Functor` data-types have a function which takes in two inputs: a function & a `functor` data-structure,

[1] See **Example 1.1** & **Example 1.2** for the Deep Embedding implementation.

and then maps the function over the data-structure.
In Haskell the *Functor* class is defined as

```
class Functor f where
  fmap :: (a->b) -> f a -> f b
```

Example 1.5 - *Functor*
Below is an example of how a *Tree* can be instanced as a *Functor*

```
instance Functor Tree where
  fmap f (Leaf x)   = Leaf (f x)
  fmap f (Fork l r) = Fork (fmap f l) (fmap f r)
```

Here if the passed value is a *Leaf* then *f* is applied to its value & returned.
If the passed value is a *Fork* then *f* is applied to the data-structures below the given *Fork*.

Remark 1.3 - *Recursiveless Datatypes*
Instead of defining data-structures which are defined recursively, in terms of themselves, we can add a parameter that shows where we the recursion would have been & now tells us the results of previous operations on the data-structure.
Doing this breaks data-types down into their smaller constituent parts which are then easier to stitch together.

Example 1.6 - *Recursiveless Datatypes*
Below is a standard definition of a *List*, without syntactic sugar

```
data List a = Empty
            | Cons a (List a)
```

Here we want to change definition of *Cons* to not include *(List a)*.
We take an additional parameter for the data-structure which replaces the recursion

```
data ListF a k = EmptyF
               | ConsF a k
```

k here tells us results of previous operations on the data-structure.
<u>N.B.</u> - We use renamed the structure to *ListF* to show it can be defined as a functor.

Definition 1.8 - ***Fix*** - *Fixed Point of Types*
The *Fixed Point* of a function is a parameter that when given to the function returns the same value.
The *Fix* data-type isolates recursion of all functors to one location, thus generalising recursion.
Fix takes a functor without its *k* as a parameter and then stores a copy of itself where the recursive *k* would go, for future operations to use.

```
data Fix f = In (f (Fix f))
```

<u>N.B.</u> - *In :: f (Fix f) → Fix f.*
<u>N.B.</u> - Considering the above example *Fix* is defined st

$$\textit{Fix (ListF a)} = \textit{In (ListF a (Fix (ListF a)))} \cong \textit{List a}$$

Example 1.7 - *Fix ListF with 2 Elements*
Below is how we define a list with 2 elements using *Fix*

$$\textit{In (ConsF 6 (In (ConsF 4 (In EmptyF))))}$$

Definition 1.9 - *inop*
inop is a function that unwraps a *Fix*

```
inop :: Fix f -> f (Fix f)
inop (In x) = x
```

Definition 1.10 - *Folds*
Folds are functions that unwrap data-structures in order to apply a given function to it.
Foldr is a particular example of a *Fold* which unwraps an *array* left-to-right.
foldr takes three inputs: a base case; a function; &, a data-structure to apply it to.
The *base case* is returned when the *"Bottom"* of the structure has been reached.

```
foldr :: (a->b->b) -> b -> [a] -> b
foldr f k []     = k
foldr f k (x:xs) = f x (foldr f k xs)
```

N.B. - This is *Foldl* as well.

Definition 1.11 - *cata* - *Catamorphism*
Catamorphisms are a generalisation of *folds*.
Algebras are functions that correspond to replacing the constructors of a data-type with functions.
cata is a function that applies an algebra to the fixpoint of a value

```
 cata :: Functor f => (f b -> b) -> Fix f -> b
 cata alg      x  = (alg . fmap (cata alg) . inop)x
≡cata alg (In x) = alg (fmap (cata alg) x)
  where
    alg :: f b -> b
```

Example 1.8 - *cata*
Below is an example of how *cata* can be implemented to converted a *Fix (ListF a)* to *List a*.

```
toList :: Fix (ListF a) -> List a
toList x = cata alg x
  where
    alg :: ListF a (List a) -> List a
    alg EmptyF       = Empty
    alg (ConsF x xs) = Cons x xs
```

N.B. - *xs* denotes the *List a* already generated.
Below is an example of how *cata* can be implemented to find the length of a *Fix(ListF a)*

```
toList :: Fix (ListF a) -> Int
toList x = cata alg x
  where
    alg :: ListF a Int -> Int
    alg EmptyF       = 0
    alg (ConsF x y)  = 1 + y
```

N.B. - *y* marks the length calculated to this point.

1.2 Parsers

This subsection was covered in Teaching Block 1; Lecture 7-13.

1.2.1 Composition

Definition 1.12 - $:+:$ - *Co-Product of Functors*
The *Co-Product of Functors* is used to extend signature functors by composing two functors into one.
The *Co-Product of Functors* is defined as

```
data (f:+:g) a = L (f a)
               | R (g a)
```

This uses two functors, f & g, to create a single functor $f:+:g$.
The *Functor* instance of *:+:* is given as

```
instance (Functor f, Functor g) => Functor (f:+:g)
  where
    fmap f (L x) = L (fmap f x)
    fmap f (R y) = R (fmap f y)
```

Example 1.9 - *Co-Product of Functors*
Consider the following functor identity for addition.

```
data ExprF k = ValF Int
             | AddF k k
instance Functor ExprF
  where
    fmap f (ValF n) = ValF n
    fmap f (AddF x y) = AddF (f x) (f y)
```

Suppose we want to end to end this include multiplication.
We define a multiplication functor identity

```
data MulF k = MulF k k
instance Functor MulF
  where
    fmap f (MulF x y) = MulF (f x) (f y)
```

We compose `ExprF` & `MulF` simply by using

```
Fix (ExprF:+:MulF)
```

This is essential the same as

```
data ExprF = ValF Int
           | AddF ExprF ExprF
           | MulF ExprF ExprF
```

Definition 1.13 - ∇ - *Junction of Algebras*
Junction of Algebras is used to compose two algebras together.
The *Junction of Algebras* is defined as

```
(∇)::(f a->a)->(g a->a)->((f:+:g)a->a)
(flag ∇ galg) (L x) = falg x
(flag ∇ galg) (R y) = falg y
```

Example 1.10 - *Junction of Algebras*
Consider the language defined in **Example 2.1**.
We define the following algebras for addition & multiplication.

```
add :: ExprF Int→Int
add (ValF x)   =x
add (AddF x y)=x + y

mul::MulF Int→Int
mul (MulF x y) = x∗y
```

We can now evaluate `ExprF`, using these semantics, using

```
eval :: Fix (ExprF :+: MulF)→Int
eval x = cata (add∇mul) x
```

1.2.2 Grammars

Definition 1.14 - *Backus-Naur Form*
Backus-Naur Form is a language used to express the shape of grammars.
Backus-Naur Form statements use the following symbols

ε	Empty Strings
$< n >$	A non-terminal
"x"	A terminal
$p\|q$	A choice between p & 1
$[e]$	Optional term
(e)	Group Terms
$e*$	0 or more of term e
$e+$	1 or more of term e

N.B. - $e+$ is written *somee* & $e*$ is written *manye*.

Example 1.11 - *Backus-Naur Form*
Below is the definition of a digit & a number using *Backus-Naur Form*

```
<digit> ::= ‘‘0”|‘‘1”|...|‘‘9”
<num>   ::= <digit> | <digit> <num>
```

Definition 1.15 - *Paull's Modified Algorithm*
Paull's Modified Algorithm is used to remove recursion from a grammar.
Consider a the following grammar

```
A ::=Aα1|...|Aαn|β1|\dots|βm
```

Where `A` is non-terminal & α_i, β_J are *Backus-Naur Form* expressions.
Paull's Modified Algorithm states to rewrite this grammar as

```
A ::=β1A'|...|βmA'
A'::=α1A'|...|αnA'|ε
```

Example 1.12 - *Paull's Modified Algorithm*
Consider a grammar for addition expressions

```
<expr>::=<expr> ‘‘+” <expr> | <num>
```

Here `A`=`<expr>`, α=`"+" <expr>` & β=`<num>`.
Applying *Paull's Modified Algorithm* we get

```
<expr> ::=<num><expr'>
<expr'>::=‘‘+”<expr><expr'> | ε
```

1.2.3 Parsers

Definition 1.16 - *Parser*
A *Parser* is a function that takes in a list of characters & returns an array of parsed data & unconsumed strings.
We can define a *Parser* type as

```
newtype Parser a = Parser (String→[(a,String)])
```

Proposition 1.1 - *Functor Instance of Parser*
Below is a definition for a `Functor` instance for `Parser`.
This allows us to transform *Parsers* into using different datatypes

```
instance Functor Parser where
  fmap f (Parser px) = Parser (λ s→[(f x,s') | (x,s')←px s])
```

Definition 1.17 - *`parse`*
In Haskell we define the function `parse` which takes in a `Parser`& `String`, then applies the parser to the string.
`parse` is defined by

```
parse :: Parser a→String→[(a,String)]
parse (Parser px) s = px s
```

Definition 1.18 - *Trivial Parsers*
There are two trivial parsers: `fail`which always fails; and, `item` which parses the first `Char` off the string.
These parsers are defined below

```
fail :: Parser a
fail = Parser (λ s→[])
item :: Parser Char
item = Parser (f)
  where
    f []     = []
    f (s:ss) = [(s,ss)]
```

Example 1.13 - *`fail` & `item`*
Here are examples of `fail` & `item` parsing the string `Hello`

```
parse fail ``Hello" = []
parse item ``Hello" = [('H',``ello")]
```

Definition 1.19 - *`look` Parser*
The `look` parser allows you to look at the input stream without consuming it

```
look :: Parser String
look = Parser (λ s→[(s,s)])
```

Definition 1.20 - *`Monad` Class*
`Monad`s are a class of data-types which allow you to map their internal value, `>>=`.
`Monad`s have a function, `return`, which wraps a value as a `Monad`.

```
class Monad m where
  (>>=)  :: m a → (a → m b) → m b
  return :: a → m a
```

Definition 1.21 - *`Monad` instance for `Parser`*
Below is the `Monad` instance for `Parser`

```
instance Monad Parser where
  return = pure
  (Parser px) >>= f = Parser (λs→concat[parse (f x) s' | (x,s')←px s])
```

`>>=` allows the function `f` to map the value `x` parsed by `px` to a new value & producing a new `Parser`.

Definition 1.22 - *`satisfy` Parser*
The `satisfy` *parser* takes a function,`p`, which maps `Char` to `Bool`.
`satisfy` uses the `item` parser to parse the first character off a string & then applies `p` to it to decide whether to return the parsed value.

```
satisfy :: (Char→Bool)→Parser Char
satisfy p = item >>= λs→ if (p t) then (pure t) else empty
```

Example 1.14 - *Parsing a Character*
Below a `Parser` is defined which either parses an `'a'` or nothing, depending on the first character of the string.

```
parse (satisfy ('a'==)) 'abc' = [('a','bc')]
parse (satisfy ('a'==)) 'xyz' = []
```

1.2.4 Parser Combinators

Definition 1.23 - *Parser Combinator*
A *Parser Combinator* is a function that takes (several) parsers as inputs & returns a new parser.

Definition 1.24 - ⟨$⟩ - *Type Change Combinator*
⟨$⟩ is a *Parser Combinator* which takes a function that maps type a to type b & a *Parser* of type b, and then returns a *Parser* of type b.

```
(⟨$⟩)::(a→b)→Parser a→Parser b
f ⟨$⟩ px = fmap f px
```

Example 1.15 - ⟨$⟩ - *`digit` Parser*
Below is a definition that changes the `item` to parse an `Int` rather than a `Char`

```
digit :: Parser Int
digit = digitToInt ⟨$⟩ item
```

Definition 1.25 - ⟨$ - *Constant Parser*
⟨$ is a *Parser Combinator* that causes a `Parser` to always return the same value.

```
(⟨$) :: a→Parser b→Parser a
x ⟨$ py = fmap (const x) py
```

Definition 1.26 - *`skip` Parser*
Below is a definition of a parser that always returns an empty array, effectively skipping parsing the rest of the string.

```
skip :: Parser a→Parser ()
skip px = () ⟨$ px
```

Definition 1.27 - *`Applicative` Functors*
The `Applicative` *Functors Class* is the class of *Functors* which have two addition functions `pure` & ⟨∗⟩.
`pure` which wraps arbitrary values as the `Functor` & ⟨∗⟩ applies a function to the context of the functor.

```
class (Functor f) => Applicative f where
  pure :: a→f a
  (⟨*⟩) :: f (a→b)→f a→f b
```

Proposition 1.2 - *`Applicative` definition of `Parser` -* ⟨∗⟩
Below is the `Applicative` instance of `Parser`

```
instance Applicative Parser where
  pure x = Parser (λs→[(x,s)])
  (Parser pf) ⟨*⟩ (Parser px) = Parser (λs→[(f x,s")
                                         |(f,s')←pf s
                                         ,(x,s")←px s'])
```

⟨∗⟩ parses a function ,f; then parses a value ,x; & then applies the function to the value, $f\ x$.
N.B. - ⟨∗⟩ is called *Ap* for apply.

Definition 1.28 - *Reverse Ap -* ⟨∗∗⟩
The below operator preforms ⟨∗⟩ in reverse.
It parses a value, then parses a function & then applies the function to the value.

```
(Parser px) ⟨**⟩ Parser pf = Parser (λs→[(f x,s'')
                                     | (x,s')←px s
                                     , (f,s")←pf s'])
```

Definition 1.29 - *`Monoidal` Class*
The `Monoidal` *Class* is the class of datatypes which have a neutral value & a function for appending the two together.

```
class Monoidal f where
  unit :: f ()
  mult :: f a → f b → f(a,b)
```

N.B. - The `Monoidal` is equivalent to `Applicative`.

Proposition 1.3 - *`Monoidal` definition of `Parser`*
Below is the `Monoidal` instance of `Parser`

```
instance Monoidal Parser where
  unit = Parser (λs→[((),s)])
  mult px py = Parser (λs→[((x,y),s")
                          | (x,s')←px s
                          | (y,s")←py s'])
```

`unit` is a `Parser` that parses noting.
`mult` is a `Parser` that takes a string & two other `Parser`s, then applies the first `Parser` to the string & then applies the second parser to the remainder of the string. `mult` returns a tuple of both these values.

Definition 1.30 - ⟨∼⟩ *- `mult` Combinator*
It is useful to define the `mult` combinator from the `Monoidal` instance of `Parser` as its own binary operation

```
(⟨~⟩) :: Monoidal f => f a → f b → f (a,b)
px ⟨~⟩ py = mult px py
```

Definition 1.31 - ⟨∗ *&* ∗⟩ *- Extracting values from* ⟨∼⟩
The following combinators are defined to extract the first or second values from the parsed tuple from ⟨∼⟩

```
(⟨*) :: Monoidal f => f a → f b → f a
px ⟨* py = fst ⟨$⟩ (px ⟨~⟩)
(*⟩) :: Monoidal f => f a → f b → f b
px *⟩ py = snd ⟨$⟩ (px ⟨~⟩)
  where
    fst (x,y) = x
    snd (x,y) = y
```

N.B. - There two additional operators which are equivalent to these ⟨∼= ⟨∗ & ∼⟩ = ∗⟩.

Definition 1.32 - `Alternative` *Class*
The `Alternative` *Class* is a subclass of `Applicative` of data types which have a neutral element,`empty` & an operator, ⟨|⟩, for choosing between two of the same type.

```
class (Applicative f) => Alternative f where
  empty :: f a
  (⟨|⟩)  :: f a → f a → f a
```

Definition 1.33 - `Alternative` *definition of* `Parser` - ⟨|⟩
Below is the `Alternative` instance of `Parser`

```
instance Alternative Parser where
  empty = fail
  (Parser px) ⟨|⟩ (Parser py) = Parser (λs→(px s)++ (py s))
```

⟨|⟩ is a combinator which returns a single array contain the result of parsing the same string with two different parsers.
N.B. - `px` & `py` must output the same data type from parsing.

Definition 1.34 - `choice`
`choice` is a function that extends the ⟨|⟩ combinator to allow for more than two parsers as input

```
choice :: [Parser a] → Parser a
choice pxs = foldr (⟨|⟩) empty pxs
```

Definition 1.35 - ⟨:⟩ - *Append Combinator*
⟨:⟩ is a combinator that appends the result of using one `Parser` onto the results of using other `Parsers`

```
(⟨:⟩) :: Parser a → Parser [a] → Parser [a]
px ⟨:⟩ pxs = (:) ⟨$⟩ px ⟨*⟩ pxs
```

Definition 1.36 - `some`
The `some` operation is equivalent to $e+$ from *Backus-Naur Form.*

```
some :: Parser a → Parser [a]
some px = px ⟨:⟩ many px
```

This parses one `Parser`, `px`, and appends this result to the result of `many px`.

Definition 1.37 - `many`
The `many` operation is equivalent to $e*$ from *Backus-Naur Form.*

```
many :: Parser a → Parser [a]
many px = some px  ⟨|⟩ pure
```

This returns the result of `some px`, if no such result exists then it returns `empty`

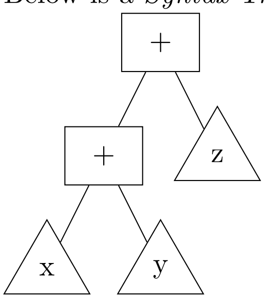
Remark 1.4 - *Summary of Combinator Operations*

$\langle\$\rangle$	Changes type of a `Parser`.
$\langle\$$	Makes a `Parser` always return the same value.
$\langle*\rangle$	Uses one `Parser` to parse a function & another `Parser` to parse a value, then applies the function to the value.
$\langle*\rangle$	Uses one `Parser` to parse a value& another `Parser` to parse a function, then applies the function to the value.
$\langle\sim\rangle$	Uses one `Parser` to parse a string& another `Parser` to parse the remaining string, then returns a tuple of the two parsed value & the remaining string.
$\langle\sim$ or $\langle*$	Performs $\langle\sim\rangle$ but returns only the result of the <u>first</u> `Parser`.
$\sim\rangle$ or $*\rangle$	Performs $\langle\sim\rangle$ but returns only the result of the <u>second</u> `Parser`.
$\langle\|\rangle$	Applies two `Parsers` to the same string & returns a single array with both results.
$\langle:\rangle$	Appends the result of using one `Parser` onto the results of using multiple other `Parsers`.

1.3 Abstract Syntax

Definition 1.38 - *Syntax Tree*
A *Syntax Trees* are used to describe dependency in a syntax.
Squares are used to signify operations & triangles to signify variables.
No shape signifies a constant value.
The children of a node show the values that are dependent upon it, only an operation will have dependants.

Example 1.16 - *Syntax Tree*
Below is a *Syntax Tree* for $(x + y) + z$.



Note that both +s are denoted as operations, while x, y, z are denoted as variables.

1.3.1 The Free Monad

Definition 1.39 - *`Free` Monad*
The *Free Monad*, `Free f a`, is used to produce syntax trees who nodes operations are defined by `f` and whose variables are all of type `a`.

```
data Free f a = Var a
              | Op (f (Free f a))
```

Example 1.17 - *`Free` Monad*
The syntax tree from **Example 3.1** can be defined, using `AddF` from **Example 2.1**, as

```
Op(AddF (Op (AddF (Var "x") (Var "y"))) (Var "z"))
```

Definition 1.40 - *Generator & Algebra*
There are two stages to interpreting `Free` trees

i) *Generator.* Changing variables into a value; &,

ii) *Algebra*. Evaluating the operations.

Definition 1.41 - *`Functor` instance of `Free`*
Defining a `Functor` of instance of `Free` allows us to replace variables with their values, since we can inspect within the tree.
N.B. - This is the *Generator* stage of interpreting `Free` trees.

```
instance Functor f => Functor (Free f) where
  fmap f (Var x) = Var (f x)
  fmap f (Op op) = Op (fmap (fmap f) op
```

Definition 1.42 - *`Monad` instance of `Free`*
We define a `Monad` instance of `Free` in order to perform *substitution*

```
instance Functor f => Monad (Free f) where
  return x = Var x
  (Var x) >>= f = f x
  (Op op) >>= f = Op (fmap (>>= f) op)
```

Definition 1.43 - *`extract`*
We define the recursive function `extract` to evaluate operations within a `Free` tree.
This is the *Algebra* stage of interpreting `Free` trees.

```
extract :: Functor f => (f b-> b)->Free f b->b
extract alg (Var x) = x
extract alg (Op op) = alg (fmap (extract alg) op)
```

N.B. - We define the use of `b` here since this function takes in the result of using `Free` functor
Definition 1.44 - *`eval`*
The `eval` function combines the *Generating* & *Algebra* stages of interpreting `Free` trees, producing a single value in the end.

```
eval :: Functor f => (f b->b)->(a->b)->Free f a->b
eval alg gen t = (extract alg)·(fmap gen) t
```

1.3.2 Additional Syntax

Definition 1.45 - *`Fail`*
`Fail` is a syntax for failure

```
data FailF k = FailF
instance Functor FailF where
  fmap f FailF = FailF
```

We need to define a `Functor` instance of `Fail` to show that no computations follow a failure
Example 1.18 - *`Fail`*
Consider define a language with division in it, `DivF`.
We need to consider the case when dividing by 0. No value will be computed so we need to signify that the failure has occurred.
We do so by allowing failure in the the *Algebra*

```
alg :: DivF (Free FailF Double)->Free FailF Double
alg (DivF (Var x) (Var 0)) = Op FailF
alg (DivF (Var x) (Var y)) = Var (x/y)
alg _                      = Op FailF
```

N.B. - this algebra also fails when an incorrect `Free` shape is passed.

Definition 1.46 - *Substitution*
Substitution allows us to replace part of an expression with another expression.
The follow syntax is used to show that all occurrences of `x` in expression `e` should be replaced with a new expression `e'`.

```
e[x↦e']
```

The `>>=` operation is used for substitution. `e >>= f` binds the substitution defined by `f` to the syntax tree/expression `e`.

Definition 1.47 - *`Or`*
We define the `Or` data-type to denote when we have a choice between two operations.

```
data Or k=Or k k
instance Functor Or where
  fmap f (Or x y) = Or (f x) (f y)
```

N.B. - We define a `Functor` instance for `Or` so that it can be used in `Free` trees.

Definition 1.48 - *`once`*
We define the function `once` to make choices about which computation stored in `Or`.
`once` simply picks the first available computation.

```
once :: Free Or a→Maybe a
once t = eval alg gen t
  where
    gen x = Just x
    alg (Or Nothing y)  = y
    alg (Or (Just x) _) = Just x
```

N.B. - We define using `y` rather than `Just y` so that when both values are `Nothing`
Definition 1.49 - *`Nondet`*
Non-Deterministic computations provide a choice between two computations.
We use $p\Box q$ to denote the operation that offers a choice between p & q.

```
type Nondet a = (Fail :+: Or) a
```

This defines a non-deterministic data-type which allows for failure.
N.B. - `Fail :+: Or` means we have `(L Fail)` & `(R (Or x y))` as structures.

Example 1.19 - *`list`*
Below is define a function `list` which traverses a `Free` tree with `Nondet` operations & returns an array of all non-failed value

```
list :: Free Nondet a→[a]
list t = eval alg gen t
  where
    gen x = [x]
    alg (L Fail) = []
    alg (R (Or x y)) = x++y
```

Definition 1.50 - *`Alt`*
Alternation is an alternative way to represent *Non-Deterministic* operations, by pairing values with booleans.
The idea being we pass `True` when we want the first value& `False` when we want the second.

```
data Alt k = Alt (Bool→k)
instance Functor Alt where
  fmap f (Alt k) = Alt (f·k)
```

This allows us to redefine `Nondet`

```
type Nondet' a = (Fail :+: Alt) a
```

Example 1.20 - *`list` with new `Nondet'`*
Below we define `list` using `Nondet'`, it still collects all non-failed values

```
list :: Free Nondet' a→[a]
list t = eval alg gen t
  where
    gen x = [x]
    alg (L Fail) = []
    alg (R (Alt k)) = (k True) ++ (k False)
```

Definition 1.51 - *`State`*
A *Stateful* computation has two particular operations `Get` & `Put`

```
data State s k = Put s k
               | Get (s→k)
```

`Put s k` will put the value `s` into the state before continuing the computation `k`.
`Get f` will only continue `f` is given a variable of type `s`.

Example 1.21 - *`evalState`*
We define `evalState` to evaluate `Free` trees with `State s` operations

```
evalState :: Free (State s) a→(s→(a,s))
evalState t = eval alg gen t
  where
    gen x s = (x,s)
    alg (Put s' k) = (λs→k s')
    alg (Get k)    = (λs→k s s) // 1st s is state that generates program
                                   2nd s is state passed on to future program
```

2 Semantics

Remark 2.1 - *Semantics*
Semantics assign meaning to expressions. They can be considered to assign meaning to *Abstract Syntax Trees.*
N.B. - A *Semantic Function* can be considered as a map from syntax to semantics.

Example 2.1 - *Semantic Function*
Defined below is a function that assigns *Semantic* meaning to a binary number, by mapping it to its decimal value

$$\begin{array}{l}\mathcal{N} : Num \to \mathbb{Z} \\ \mathcal{N}[\![0]\!] = 0 \\ \mathcal{N}[\![1]\!] = 1 \\ \mathcal{N}[\![n0]\!] = 2 * \mathcal{N}[\![n]\!] \\ \mathcal{N}[\![n1]\!] = 1 + 2 * \mathcal{N}[\![n]\!]\end{array}$$

Definition 2.1 - *Program State*
A *Program State* is an injective map from variables to integers.

$$State = Var \to \mathbb{Z}$$

A *Program State* is used with a *Semantic Function* to find the result of the *Semantic Function* for different values of the variables. (The *Program State* is updated throughout the execution of the *Semantic Function*).
N.B. - See **0. Reference** for notation.

Definition 2.2 - *Semantic Equivalence*
Two statements, $a\&b$, are semantically equivalent iff $\mathcal{F}[\![a]\!]s = \mathcal{F}[\![b]\!]\ \forall\ s \in State$, for an appropriate $\mathcal{F}$.

Definition 2.3 - *Arithmetic Semantic Function*
Defined below is a *Semantic Function* for arithmetic expressions

$$\begin{array}{lcl}\mathcal{A} & : & Aexp \to (State \to \mathbb{Z}) \\ \mathcal{A}[\![n]\!]s & = & s\ x \\ \mathcal{A}[\![a_1 + a_2]\!]s & = & \mathcal{A}[\![a_1]\!]s + \mathcal{A}[\![a_2]\!]s \\ \mathcal{A}[\![a_1 - a_2]\!]s & = & \mathcal{A}[\![a_1]\!]s - \mathcal{A}[\![a_2]\!]s \\ \mathcal{A}[\![a_1 * a_2]\!]s & = & \mathcal{A}[\![a_1]\!]s * \mathcal{A}[\![a_2]\!]s\end{array}$$

Definition 2.4 - *Boolean Semantic Function*
Defined below is a *Semantic Function* for boolean expressions

$$\begin{array}{lcl}\mathcal{B} & : & Bexp \to (State \to T) \\ \mathcal{B}[\![true]\!]s & = & tt \\ \mathcal{B}[\![false]\!]s & = & ff \\ \mathcal{B}[\![a_1 = a_2]\!]s & = & \begin{cases} tt & \mathcal{A}[\![a_1]\!]s = \mathcal{A}[\![a_2]\!]s \\ ff & \mathcal{A}[\![a_1]\!]s \neq \mathcal{A}[\![a_2]\!]s \end{cases} \\ \mathcal{B}[\![a_1 \leq a_2]\!]s & = & \begin{cases} tt & \mathcal{A}[\![a_1]\!]s \leq \mathcal{A}[\![a_2]\!]s \\ ff & \mathcal{A}[\![a_1]\!]s > \mathcal{A}[\![a_2]\!]s \end{cases} \\ \mathcal{B}[\![\neg b]\!]s & = & \begin{cases} tt & \mathcal{B}[\![b]\!]s = tt \\ ff & \mathcal{B}[\![b]\!]s = ff \end{cases} \\ \mathcal{B}[\![b_1 \wedge b_2]\!]s & = & \begin{cases} tt & \mathcal{B}[\![b_1]\!]s = tt \text{ and } \mathcal{B}[\![b_2]\!]s = tt \\ ff & \mathcal{B}[\![b_1]\!]s = ff \text{ or } \mathcal{B}[\![b_2]\!]s = ff \end{cases}\end{array}$$

N.B. - $T = \{tt, ff\}$.

Definition 2.5 - *Free Variables of Semantics*
The *Free Variables* of an arithmetic expression are the set of variables occurring within it.
e.g. The free variables of $(x+y)*y$ are $\{x, y\}$.

Definition 2.6 - *Substitutions of Semantics*
Substitution in semantics is the process of replacing a variable with an expression.
Suppose we have an expression a & want to replace all occurrences of variable y with the expression a_0 we write

$$a[y \mapsto a_0]$$

Formally we define substitution as

$$\begin{array}{rcl}
n[y \mapsto a_0] & = & n \\
x[y \mapsto a_0] & = & \begin{cases} a_0 & x = y \\ x & x \neq y \end{cases} \\
(a_1 + a_2)[y \mapsto a_0) & = & (a_1[y \mapsto a_0]) + (a_2[y \mapsto a_0]) \\
(a_1 - a_2)[y \mapsto a_0) & = & (a_1[y \mapsto a_0]) - (a_2[y \mapsto a_0]) \\
(a_1 * a_2)[y \mapsto a_0) & = & (a_1[y \mapsto a_0]) * (a_2[y \mapsto a_0]) \\
(s[y \mapsto v])x & = & \begin{cases} v & x = y \\ s\ x & x \neq y \end{cases}
\end{array}$$

2.1 Operational Semantics

Remark 2.2 - *Operational Semantics*
Operational Semantics is concerned with how a computation is executed, not just the result.

Definition 2.7 - *Configuration*
There are two types of *Configuration*

i) An *intermediate configuration* - $\langle x, \sigma \rangle$ where x is a syntactic expression & σ is a state.

ii) A *Final Configuration* - y a syntactic value.

Notation 2.1 - *Derivation Tree*
Let δ_i & γ_i be configurations st $\delta_i \to \gamma_i$.
Let $\delta_0 \to \gamma_0$ be the result we want to show, this is called the *Conclusion*.
We need to derive other configurations to prove that this holds, these are called *Premises*.

$$\frac{\delta_1 \to \gamma_1 \dots \delta_n \to \gamma_n}{\delta_0 \to \gamma_0}$$

N.B. - More layers can be added when a *Premises* needs to be proved.
Remark 2.3 - *Rule v Axiom*
Rules have one conclusion & at least one premise.
Axiomata have no premises.
N.B. - In **Definition 2.8** $[skip_{ns}]$ is an *axiom* & $[comp_{ns}]$ is a *rule*.

2.1.1 Natural Operational Semantics

Remark 2.4 - *Natural Operational Semantics*
Natural Operational Semantics is concerned with the relationship between the initial state & final state.

Thus the transition relation, $\to$, specifies the relationship between initial & final state for each statement.
$\langle S, s\rangle \to s'$ means that when statement S is applied to state s it results in state s'.
N.B. - This is often referred to as the *Big Step*.

Definition 2.8 - *Rules & Axioms of Natural Semantics*

$$\begin{array}{ll}
[skip_{ns}] & \overline{\langle skip, \sigma\rangle \to \sigma} \\
[ass_{ns}] & \overline{\langle x := a\rangle \to \sigma[x \mapsto \mathcal{A}[\![a]\!]\sigma]} \\
[comp_{ns}] & \dfrac{\langle S_1, s\rangle \to s',\ \langle S_2, s'\rangle \to s''}{\langle S_1; S_2, s\rangle \to s''} \\
[if^{tt}_{ns}] & \dfrac{\langle S_1, s\rangle \to s'}{\langle if\ b\ then\ S_1\ else\ S_2, s\rangle \to s'} \text{ if } \mathcal{B}[\![b]\!]s = tt \\
[if^{ff}_{ns}] & \dfrac{\langle S_2, s\rangle \to s'}{\langle if\ b\ then\ S_1\ else\ S_2, s\rangle \to s'} \text{ if } \mathcal{B}[\![b]\!]s = ff \\
[while^{tt}_{ns}] & \dfrac{\langle S, s\rangle \to s',\ \langle while\ b\ do\ S, s'\rangle \to s''}{\langle while\ b\ do\ S, s\rangle \to s''} \text{ if } \mathcal{B}[\![b]\!] = tt \\
[while^{tt}_{ns}] & \langle while\ b\ do\ S, s\rangle \to s \text{ if } \mathcal{B}[\![b]\!] = ff
\end{array}$$

Definition 2.9 - *Semantic Function,* $\mathcal{S}_{ns}$
This *Semantic Function* is a partial function which summarises the *meaning* of *Natural Semantics* statements

$$\begin{array}{rcl}
\mathcal{S}_{ns} & : & Stm \to (State \hookleftarrow State) \\
\mathcal{S}_{ns}[\![S]\!]\sigma & = & \begin{cases} \sigma' & \langle S, \sigma\rangle \to \sigma' \\ \perp & \text{otherwise} \end{cases}
\end{array}$$

N.B. - $\mathcal{S}_{ns}[\![S]\!] = \mathcal{S}_{sos}[\![S]\!]\ \forall\ S$.

Definition 2.10 - *Types of Statement*
There are two types of statements

i) *Terminating* iff $\exists\ s'\ st\ \langle S, s\rangle \to s'$

ii) *Looping* iff $\nexists\ s'\ st\ \langle S, s\rangle \to s'$

A statement *always terminates* if it is terminating $\forall\ s$.
A statement *always loops* if it loops $\forall\ s$.

Definition 2.11 - *Equivalence*
Two statements are equivalent if

$$\forall\ s\langle S_1, s\rangle \to s'\ and\ \langle S_2, s\rangle \to s'$$

Definition 2.12 - *Deterministic*
A statement S is *Deterministic* if

$$\forall\ s, s', s''\ if\ \langle S, s\rangle \to s'\ \&\ \langle S, s\rangle \to s'' \implies s' = s''$$

2.1.2 Structural Operational Semantics

Remark 2.5 - *Structural Operation Semantics*
Structural Operation Semantics is concerned with the individual steps of execution.
The transition relation, $\Rightarrow$, expresses the first step of the execution of a statement.
$\langle S, s\rangle \Rightarrow \langle S', s'\rangle$ means that when executing the first step of statement S on state s results in statement S' & state s'.

Definition 2.13 - *Rules & Axiomata of Structural Semantics*

$[skip_{sos}]$	$\langle x := a, s\rangle \Rightarrow s[x \mapsto \mathcal{A}[\![a]\!]s]$
$[ass_{sos}]$	$\langle skip, s\rangle \Rightarrow s$
$[comp^1_{sos}]$	$\dfrac{\langle S_1, s\rangle \Rightarrow \langle S_1', s'\rangle}{\langle S_1; S_2, s\rangle \Rightarrow \langle S_1'; S_2, s'\rangle}$
$[comp^2_{sos}]$	$\dfrac{\langle S_1, s\rangle \Rightarrow s'}{\langle S_1; S_2, s\rangle \Rightarrow \langle S_2, s'\rangle}$
$[if^{tt}_{sos}]$	$\langle\ if\ b\ then\ S_1\ else\ S_2, s\rangle \Rightarrow \langle S_1, s\rangle$ if $\mathcal{B}[\![b]\!] = tt$
$[if^{ff}_{sos}]$	$\langle\ if\ b\ then\ S_1\ else\ S_2, s\rangle \Rightarrow \langle S_2, s\rangle$ if $\mathcal{B}[\![b]\!] = ff$
$[while_{sos}]$	$\langle while\ b\ do\ S, s\rangle \Rightarrow \langle if\ b\ then\ (S;\ while\ b\ do\ S)\ else\ skip, s\rangle$

Definition 2.14 - *Semantic Function, $\mathcal{S}_{sos}$*
This *Semantic Function* is a partial function which summarises the *meaning* of *Operational Semantics* statements

$$\begin{aligned}\mathcal{S}_{sos} &: Stm \to (State \hookleftarrow State)\\ \mathcal{S}_{sos}[\![S]\!]s &= \begin{cases} s' & \langle S, s\rangle \Rightarrow^* s' \\ \perp & \text{otherwise}\end{cases}\end{aligned}$$

<u>N.B.</u> - $\mathcal{S}_{ns}[\![S]\!] = \mathcal{S}_{sos}[\![S]\!]\ \forall\ S$.

Definition 2.15 - *Stuck Configuration*
A *Configuration* $\langle S, s\rangle$ is *stuck* if

$$\nexists\ S', s'\ st\ \langle S, s\rangle \Rightarrow \langle S', s'\rangle$$

Definition 2.16 - *Types of Derivation Sequence*
Let $\gamma_i = \langle S_i, s_i\rangle$ be configurations.
There are two types of *Derivation Sequence*

i) *Finite Sequence* iff $\exists\ k \in \mathbb{N}$ st γ_k is a terminating or stuck configuration of γ_0.

ii) *Infinite Sequence* if no such k exists.

<u>N.B.</u> - A configuration terminates iff it derives a finite sequence. A configuration loops iff it derives an infinite sequence.

2.1.3 Natural Operational Semantics v Structural Operational Semantics

Proposition 2.1 - *Summary of Proof that $\mathcal{S}_{ns}[\![S]\!] = \mathcal{S}_{sos}[\![S]\!]\ \forall\ S$*

i) Proof by *Induction on the Shape of Derivation Trees* that for each derivation tree in the natural semantics there is a corresponding finite derivation sequence in the structural operational semantics.

ii) Proof by *Induction on the Length of Derivation Sequences* that for each finite derivation sequence in the structural operational semantics there is a corresponding derivation tree in the natural semantics.

Proposition 2.2 - *Abnormal Termination*
In a natural semantics we cannot distribugish between looping & abnormal termination (*i.e.* `abort`.
In structural semantics looping is reflected by infinite derivation sequences and abnormal termination by finite derivation sequences ending in a stuck configuration.

Proposition 2.3 - *Non-Determinism*
In a natural semantics non-determinism will suppress looping, if possible.

In a structural semantics non-determinism does not suppress looping.

Proposition 2.4 - *Parallelism*
In a natural semantics the execution of the immediate constituents is atomic entity so we cannot express interleaving computations.
In a structural operation semantics we concentrate on the small steps of the computation, so can easily express interleaving.

2.2 Provably Correct Implementation

Remark 2.6 - *Correctness of a Language*
Defining a formal specification for the semantics of a language using one the semantic forms defined in this unit allows us to prove the correctness of its implementation.
Here the correctness of *While* is proved by define an *Operational Semantics* for it, which can then be executed by an *Abstract Machine*.

Proposition 2.5 - *Process of Proving Correctness*
The correctness result states that if we can

- Translate a program into code;
- Execute the code on the abstract machine; &
- Get the same result as specified by the semantic functions S_{ns} & S_{sos}

then the language is correct.

Definition 2.17 - *Abstract Machine*
An *Abstract Machine* is a theoretical model of a computer.
*Abstract Machine*s use configurations of the form $\langle c, e, s\rangle$ where

- c is the code to be executed;
- e is the evaluation stack which evaluates arithmetic & boolean expression; and
- s is the storage of values for variables.

Definition 2.18 - *Instructions for Abstract Machine*
The *Instructions* for the *Abstract Machine* are given by the following abstract syntax

inst	::=	PUSH-n |ADD |MULT |SUB
	|	TRUE |FALSE |EQ |LE |AND |NEG
	|	FETCH-x |STORE-x
	|	NOOP |BRANCH(c, c) |LOOP(c, c)
c	::=	ε |inst:c

Definition 2.19 - *Terminal Configuration*
A *Configuration*, $\langle c, e, s\rangle$ is *terminal* if it has the form $\langle \varepsilon, e, s\rangle$ (*i.e.* It's code component is empty).

Definition 2.20 - *Transition Relation,* $\triangleright$
The transition relation $\triangleright$ shows the result of *one-step* of execution.$\langle c, e, s\rangle \triangleright \langle c', e', s'\rangle$ means $\langle c, e, s\rangle$ becomes $\langle c', e', s'\rangle$ after executing the first expression in the stack.

Definition 2.21 - *Definitions of Instructions*

$\langle \text{PUSH-}n : c, e, s\rangle$	$\triangleright$	$\langle c, \mathcal{N}[[n]] : e, s\rangle$	
$\langle \text{ADD: } c, z_1 : z_2 : e, s\rangle$	$\triangleright$	$\langle c, (z_1 + z_2) : e, s\rangle$	if $z_1, z_2 \in \mathbb{Z}$
$\langle \text{MULT: } c, z_1 : z_2 : e, s\rangle$	$\triangleright$	$\langle c, (z_1 \times z_2) : e, s\rangle$	if $z_1, z_2 \in \mathbb{Z}$
$\langle \text{SUB: } c, z_1 : z_2 : e, s\rangle$	$\triangleright$	$\langle c, (z_1 - z_2) : e, s\rangle$	if $z_1, z_2 \in \mathbb{Z}$
$\langle \text{TRUE: } c, e, s\rangle$	$\triangleright$	$\langle c, tt : e, s\rangle$	
$\langle \text{FALSE: } c, e, s\rangle$	$\triangleright$	$\langle c, ff : e, s\rangle$	
$\langle \text{EQ: } c, z_1 : z_2 : e, s\rangle$	$\triangleright$	$\langle c, (z_1 = z_2) : e, s\rangle$	if $z_1, z_2 \in \mathbb{Z}$
$\langle \text{LE: } c, z_1 : z_2 : e, s\rangle$	$\triangleright$	$\langle c, (z_1 \leq z_2) : e, s\rangle$	if $z_1, z_2 \in \mathbb{Z}$
$\langle \text{AND: } c, t_1 : t_2 : e, s\rangle$	$\triangleright$	$\langle c, (t_1 \wedge t_2) : e, s\rangle$	if $z_1, z_2 \in \mathbb{Z}$
$\langle \text{NEG: } c, t_1 : t_2 : e, s\rangle$	$\triangleright$	$\langle c, (t_1 \neq t_2) : e, s\rangle$	if $z_1, z_2 \in \mathbb{Z}$
$\langle \text{FETCH}-x : c, e, s\rangle$	$\triangleright$	$\langle c, (s\ x) : e, s\rangle$	
$\langle \text{STORE}-x : c, z : e, s\rangle$	$\triangleright$	$\langle c, e, s[x \mapsto z]\rangle$	if $z \in \mathbb{Z}$
$\langle \text{BRANCH}(c_1, c_2) - x : c, t : e, s\rangle$	$\triangleright$	$\begin{cases} \langle c_1 : c, e, s\rangle & if\ t = tt \\ \langle c_2 : c, e, s\rangle & if\ t = ff \end{cases}$	
$\langle \text{LOOP}(c_1, c_2) : c, e, s\rangle$	$\triangleright$	$\langle c_1 :\text{BRANCH}(c_2 :\text{LOOP}(c_1, c_2),\text{NOOP}) : c, e, s\rangle$	

Definition 2.22 - *Execution Function*

We define meaning for a sequence of instructions by mapping to a partial function from State to State.

$$\begin{array}{rcl} \mathcal{M} & : & \text{Code}\to(\text{State}\hookrightarrow\text{State}) \\ \mathcal{M}[[c]]\sigma & = & \begin{cases} \sigma' & if\ \langle c, \varepsilon, \sigma\rangle \triangleright^* \langle \varepsilon, e, \sigma'\rangle \\ \text{Undefined} & \text{Otherwise} \end{cases} \end{array}$$

Proposition 2.6 - *Generating Code for Abstract Machine*

We define total functions from *While* language constructs to *Abstract Machine* code, to generate code. Below are examples for *Aexp*, *Bexp* & *Stm*

$$\begin{array}{lcl} \mathcal{CA} & : & \text{Aexp}\to\text{Code} \\ \mathcal{CA}[[n]] & = & \text{PUSH-}n \\ \mathcal{CA}[[x]] & = & \text{FETCH-}n \\ \mathcal{CA}[[a_1 + a_2]] & = & \mathcal{CA}[[a_2]] : \mathcal{CA}[[a_1]]\text{:ADD} \\ \mathcal{CA}[[a_1 \times a_2]] & = & \mathcal{CA}[[a_2]] : \mathcal{CA}[[a_1]]\text{:MULT} \\ \mathcal{CA}[[a_1 - a_2]] & = & \mathcal{CA}[[a_2]] : \mathcal{CA}[[a_1]]\text{:SUB} \end{array}$$

$$\begin{array}{lcl} \mathcal{CB} & : & \text{Bexp}\to\text{Code} \\ \mathcal{CB}[[\text{true}]] & = & \text{TRUE} \\ \mathcal{CB}[[\text{false}]] & = & \text{FALSE} \\ \mathcal{CB}[[a_1 = a_2]] & = & \mathcal{CA}[[a_2]] : \mathcal{CA}[[a_1]]\text{:EQ} \\ \mathcal{CB}[[a_1 \leq a_2]] & = & \mathcal{CA}[[a_2]] : \mathcal{CA}[[a_1]]\text{:LE} \\ \mathcal{CB}[[\neg b]] & = & \mathcal{CB}[[b]]\text{:NEG} \\ \mathcal{CB}[[b_1 \wedge b_2]] & = & \mathcal{CB}[[b_2]] : \mathcal{CB}[[b_1]]\text{:AND} \end{array}$$

$$\begin{array}{lcl} \mathcal{CS} & : & \text{Stm}\to\text{Code} \\ \mathcal{CS}[\![x := a]\!] & = & \mathcal{CA}[\![a]\!]\text{:STORE-}x \\ \mathcal{CS}[\![\text{skip}]\!] & = & \text{NOOP} \\ \mathcal{CS}[\![S_1 : S_2]\!] & = & \mathcal{CS}[\![S_1]\!] : \mathcal{CS}[\![S_2]\!] \\ \mathcal{CS}[\![\text{if } b \text{ then } S_1 \text{ else } S_2]\!] & = & \mathcal{CB}[\![b]\!]\text{:BRANCH}(\mathcal{CS}[\![S_1]\!], \mathcal{CS}[\![S_2]\!]) \\ \mathcal{CS}[\![\text{while } b \text{ do } S]\!] & = & \text{LOOP}(\mathcal{CB}[\![b]\!], \mathcal{CS}[\![S]\!]) \end{array}$$

Definition 2.23 - *Semantic Function, $\mathcal{S}_{am}$*

This *Semantic Function* is a partial function that obtains meaning for a statement by translating it into code for the *Abstract Machine* and then executing the code in the *Abstract Machine*

$$\begin{array}{rcl}\mathcal{S}_{am} & : & Stm \rightarrow (State \hookrightarrow State) \\ \mathcal{S}_{am}[\![S]\!] & = & \mathcal{M}(\mathcal{CS}[\![S]\!])\end{array}$$

Proposition 2.7 - *Summary of Proof for Correctness of Implementation of While Language*

i) Prove by *Induction on the Shape of Derivation Trees* that for each derivation tree in the natural semantics there is a corresponding finite computation sequence on the abstract machine.

ii) Prove by *Induction on the Length of Computation Sequences* that for each finite computation sequence obtained from executing a statement of *While* on the abstract machine there is a corresponding derivation tree in the natural semantics.

2.3 Denotational Semantics

Remark 2.7 - *Denotational Semantics*
Denotational Semantics is concerned with the association between the initial state & final state of a computation. *Denotational Semantics* defines a *Semantic Function* for each *Syntactic Category* which map a syntactic construct to a mathematical object, generally a function, that describes the effect of executing that construct.
Theorem2.1 - *Fixed-Point Theorem*
Let $f : D \rightarrow D$ be a continuous function on a *Chain-Complete Partially Ordered Set* $(D, \sqsubseteq)$ with least element $\bot$. Then

$$FIX\ f = \sqcup\{f^n(\bot)|n \in \mathbb{N} \in D\} \text{ exists \& is the least} - \text{fixpoint of } f$$

N.B. - $f^0 = id$ & $f^n = f(f^{n-1})$.

2.3.1 Direct Denotational Semantics

Definition 2.24 - *Conditional Function, cond*
The conditional function takes in three functions, a boolean & two state maps. When applied to a state it uses the boolean function to decide which map to use.

$$\begin{array}{rcl} cond & : & (State \rightarrow T) \times (State \hookrightarrow State) \times (State \hookrightarrow State) \rightarrow (State \hookrightarrow State) \\ cond(b, c, d)x & = & \begin{cases} c(x) & \text{if } b(x) = tt \\ d(x) & \text{otherwise} \end{cases}\end{array}$$

Definition 2.25 - *Semantic Function,* $\mathcal{S}_{ds}$
This *Semantic Function* is a partial function which summarises the *meaning* of *Direct Semantics* statements

$$\begin{array}{lcl} \mathcal{S}_{ds} & : & Stm \rightarrow (State \hookrightarrow State) \\ \mathcal{S}_{ds}[\![x := a]\!]s & = & s[\![x \mapsto \mathcal{A}[\![a]\!]s] \\ \mathcal{S}_{ds}[\![skip]\!] & = & id \\ \mathcal{S}_{ds}[\![S_1; S_2]\!] & = & \mathcal{S}_{ds}[\![S_2]\!] \cdot \mathcal{S}_{ds}[\![S_1]\!] \\ \mathcal{S}_{ds}[\![if\ b\ then\ S_1\ else\ S_2]\!] & = & cond(\mathcal{B}[\![b]\!], \mathcal{S}_{ds}[\![S_1]\!], \mathcal{S}_{ds}[\![S_2]\!]) \\ \mathcal{S}_{ds}[\![while\ b\ do\ S]\!] & = & FIX\ F \text{ where } Fg = cond(\mathcal{B}[\![b]\!], g \cdot \mathcal{S}_{ds}[\![S]\!], id) \end{array}$$

N.B. - FIX is the fixpoint operator & *cond* is the conditional function.

Proposition 2.8 - *Proof that Denotational Semantics is Well Defined*

i) The set $State \hookrightarrow State$ equipped with an appropriate order $\sqsubset$ is a chain-complete partially ordered set.

ii) Certain functions $\Psi : (State \hookrightarrow State) \to (State \hookrightarrow State)$ are continuous.

iii) In the definition of $\mathcal{S}_{ds}$ we only apply the fixed point operation to continuous functions.

Theorem2.2 - $\mathcal{S}_{sos}[\![S]\!] = \mathcal{S}_{ds}[\![S]\!]$
Further

$$\mathcal{S}_{sos}[\![S]\!] \sqsubseteq \mathcal{S}_{ds}[\![S]\!] \ \forall\ S$$

and

$$\mathcal{S}_{ds}[\![S]\!] \sqsubseteq \mathcal{S}_{sos}[\![S]\!] \ \forall\ S$$

Proposition 2.9 - *Summary of Proof for Equivalence of Operational & Denotational Semantics*

i) Prove that $\mathcal{S}_{sos}[\![S]\!] \sqsubseteq \mathcal{S}_{ds}[\![S]\!]$ by first using *Induction on the Shape of Derivation Trees* to show that

   - If one step of a statement is executed in the structural operational semantics and does not terminate then this does not change the meaning in the denotational semantics; and,

   Then, secondly, using *Induction on the Length of Derivation Sequences* show that

   - If one step of a statement is executed in the structural operational semantics and does terminate, then the same result is obtained in the denotational semantics.

ii) Prove that $\mathcal{S}_{ds}[\![S]\!] \sqsubseteq \mathcal{S}_{sos}[\![S]\!]$ by showing that

   - $\mathcal{S}_{sos}$ fulfils slightly weaker versions of the clauses defining $\mathcal{S}_{ds}$.

   A proof by *Structural Induction* gives that $\mathcal{S}_{ds}[\![S]\!] \sqsubseteq \mathcal{S}_{sos}[\![S]\!]$

2.3.2 Continuation Denotational Semantics

I think this is an extension to the *While* language.

$$\begin{array}{lcl}
\mathcal{S}_{ds} & : & Stm \to (State \hookrightarrow State) \\
\mathcal{S}_{cs}[\![x := a]\!]\ c\ s & = & c(s[x \mapsto \mathcal{A}[\![a]\!]s) \\
\mathcal{S}_{cs}[\![skip]\!] & = & id \\
\mathcal{S}_{cs}[\![S_1; S_2]\!] & = & \mathcal{S}_{cs}[\![S_1]\!] \cdot \mathcal{S}_{cs}[\![S_2]\!] \\
\mathcal{S}_{cs}[\![if\ b\ then\ S_1\ else\ S_2]\!]c & = & cond(\mathcal{B}[\![b]\!], \mathcal{S}_{cs}[\![S_1]\!]c, \mathcal{S}_{cs}[\![S_2]\!]c) \\
\mathcal{S}_{cs}[\![while\ b\ do\ S]\!] & = & FIX\ G \text{ where } (G\ g)c = cond(\mathcal{B}[\![b]\!], \mathcal{S}_{cs}[\![S]\!](g\ c), c)
\end{array}$$

2.4 Axiomatic Semantics

Definition 2.26 - *Correctness Properties*
There are two types of *Correctness Properties*

i) *Partial Correctness Property* - States that <u>if</u> the program terminates then a certain relationship between initial & final state will hold.

ii) *Total* - States the program <u>will</u> terminate & when it does then a certain relationship between initial & final state will hold.

Remark 2.8 - *Axiomatic Semantics*
An *Axiomatic Semantics* allows us to prove a program satisfies a *partial* or *total correctness property.*

Proposition 2.10 - *Assertion Triple*
We define *assertions* as a triple of the form

$$\text{Pre-Condition} \quad \text{Program} \quad \text{Post-Condition}$$

The *Pre* & *Post-Conditions* are *Predicate functions* (*i.e.* If s holds for the pre-condition & when the program is executed on s it produces s' then s' holds for the post-condition.
N.B. - These are know as *Hoare Triples.*

Definition 2.27 - *Assertion Language*
The following notation is used for defining for complex *Predicates*

$P = P_1 \land P_2$	where	$P\ s = P_1\ s$ and $P\ s = P_2\ s\ \forall\ s$
$P = P_1 \lor P_2$	where	$P\ s = P_1\ s$ or $P\ s = P_2\ s\ \forall\ s$
$P = \neg P'$	where	$P\ s = \neg(P'\ s)\ \forall\ s$
$P = P'[x \mapsto \mathcal{A}[\![a]\!]]$	where	$P\ s = P'(s[x \mapsto \mathcal{A}[\![a]\!]])$
$P \implies P'$	where	$P\ s \implies P'\ \forall\ s$

Definition 2.28 - *Logical Variables*
Logical Variables are variables used in the *Pre* & *Post-Conditions* of an assertion, but not in the *Program. Logical Variables* are used to remember initial values.

Definition 2.29 - *Program Variables*
Program Variables are variables used in the *Program* of an assertion & thus appear in the state of the *Program.*

Definition 2.30 - *Inference Tree*
An *Inference Tree* is analogous to a *Derivation Tree* except they are used to show how to infer a property. The leaves (top layers) of an *Inference Tree*s are axioms and the internal nodes are rules.
N.B. - An *Inference Tree* is called *Simple* if it is an instance of one of the axioms, otherwise it is called *Composite.*

Definition 2.31 - *Provable Equivalence of Statements*
The programs S_1 & S_2 are *Provably Equivalent* iff

$$\forall\ P, Q\ \vdash\ \{P\}S_1\ \{Q\} \Leftrightarrow \vdash\ \{P\}S_2\ \{Q\}$$

2.4.1 Partial Axiomatic Semantics

Remark 2.9 - *Partial Correctness Assertions*
Partial Correctness Assertions prove a *Partial Correctness Property* by just considering the *essential properties* of constructs. This is done by defining *assertions* about properties of a program. We denote these assertions as

$$\{P\}\ S\ \{R\}$$

This means P holds before S is executed & if S terminates then R will hold.

Definition 2.32 - *Axiomatic System for Partial Correctness*

$[ass_p]$	$\{P[x \mapsto \mathcal{A}[\![a]\!]]\}$ x:=a $\{P\}$
$[skip_p]$	$\{P\}$ skip $\{P\}$
$[comp_p]$	$\dfrac{\{P\}\ S_1\ \{Q\},\quad \{Q\}\ S_2\ \{R\}}{\{P\}\ S_1; S_2\ \{R\}}$
$[if_p]$	$\dfrac{\{\mathcal{B}[\![b]\!] \land P\}\ S_1\ \{Q\},\quad \{\neg\mathcal{B}[\![b]\!] \land P\}\ S_2\ \{Q\}}{\{P\}\ if\ b\ then\ S_1\ else\ S_2\ \{Q\}}$
$[while_p]$	$\dfrac{\{\mathcal{B}[\![b]\!] \land P\}\ S\ \{P\}}{\{P\}\ while\ b\ do\ s\ \{\neg\mathcal{B}[\![b]\!] \land P\}}$
$[cons_p]$	$\dfrac{\{P'\}\ S\ \{Q'\}}{\{P\}\ S\ \{Q\}}$ if $P \implies P'$ & $Q' \implies Q$.

Definition 2.33 - *Weakest Liberal Precondition*
We define the *Weakest Liberal Precondition* st

$$wlp(S,Q)\ s = tt \text{ iff } \forall\ s'\ \langle S, s\rangle \to s' \implies Q\ s' = tt$$

2.4.2 Total Axiomatic Semantics

Remark 2.10 - *Total Correctness Assertion*
Total Correctness Assertion prove a *Total Correctness Property.* We denote these assertion as

$$[P]\ S\ [Q]$$

This means that P holds before S is executed, S will terminate & when it does Q will hold.

Definition 2.34 - *Axiomatic System for Total Correctness*

$[ass_t]$	$\big[P[x \mapsto \mathcal{A}[\![a]\!]]\big]\ x := a\ [P]$
$[skip_t]$	$[P]\ skip\ [P]$
$[comp_t]$	$\dfrac{[P]\ S_1\ [Q],\quad [Q]\ S_2\ [R]}{[P]\ S_1; S_2\ [R]}$
$[if_t]$	$\dfrac{[\mathcal{B}[\![b]\!] \land P]\ S_1\ [Q],\quad [\neg\mathcal{B}[\![b]\!] \land P]\ S_2\ [Q]}{[P]\ if\ b\ then\ S_1\ else\ S_2\ [Q]}$
$[while_t]$	$\dfrac{[P(z+1)]\ S\ [P(z)]}{\exists\ z \in \mathbb{N}\ P(z)]\ while\ b\ do\ S\ [P(0)]}$ where $P(z+1) \implies \mathcal{B}[\![b]\!]$ & $P(0) \implies \neg\mathcal{B}[\![b]\!]$
$[cons_t]$	$\dfrac{[P']\ S\ [Q]'}{[P]\ S\ [Q]}$ where $P \implies P'$ & $Q \implies Q'$

0 Reference

0.1 While Language

The *While* language is a simple imperative language used for examples of semantics.

0.1.1 Base

The *While* language has the following *syntactic categories*

n	Numerals (See *Num*)
x	Variables (See *Var*)
a	Arithmetic Expressions (See *Aexp*)
b	Boolean Expressions(See *Bexp*)
S	Statements (See *Stm*)

N.B. - Subscripts, x_1, & primes, x' are used to differentiate variables of the same syntactic class.

The *While* language has the following *abstract syntax*

Aexp	::=	Num	Bexp	::=	'true'	Stm	::=	$x := a$
	|	Var		|	'false'		|	'skip'
	|	$a_1 + a_2$		|	$a_1 = a_2$		|	$S_1; S_2$
	|	$a_1 * a_2$		|	$a_1 \leq a_2$		|	if b then S_1 else S_2
	|	$a_1 - a_2$		|	$\neg b$		|	while b do S
				|	$b_1 \wedge b_2$		|	'abort'
							|	S_1 or S_2
							|	S_1 par S_2
							|	begin D_V D_p S end

N.B. - 'true' & 'false' are called basis elements. Constructs involving other expressions are called composites.

'`abort`' stops execution of a program.

'S_1 `or` S_2' enables *non-determinism.*
The natural semantics of 'S_1 `or` S_2' is defined as

$$[or^1_{ns}] \quad \frac{\langle S_1, s\rangle \to s'}{\langle S_1 \ or \ S_2, s\rangle \to s'}$$

$$[or^2_{ns}] \quad \frac{\langle S_2, s\rangle \to s'}{\langle S_1 \ or \ S_2, s\rangle \to s'}$$

The structural semantics of 'S_1 `or` S_2' is defined as

$$[or^1_{sos}] \quad \langle S_1 \ or \ S_2, s\rangle \Rightarrow \langle S_1, s\rangle$$

$$[or^2_{sos}] \quad \langle S_1 \ or \ S_2, s\rangle \Rightarrow \langle S_2, s\rangle$$

'S_1 `par` S_2' enables *Parallelism* where S_1 & S_2 have to be executed, but that the execution can be interleaved.
The natural semantics of 'S_1 `par` S_2' is defined as

$$[par^1_{ns}] \quad \frac{\langle S_1, s\rangle \to s',\ \langle S_2, s'\rangle \to s''}{\langle S_1 \ par \ S_2, s\rangle \to s''}$$

$$[par^2_{ns}] \quad \frac{\langle S_2, s\rangle \to s',\ \langle S_1, s'\rangle \to s''}{\langle S_1 \ par \ S_2, s\rangle \to s''}$$

The structural semantics of 'S_1 `par` S_2' is defined as

$$[par^1_{sos}] \quad \frac{\langle S_1, s\rangle \Rightarrow \langle S_1', s'\rangle}{\langle S_1\ par\ S_2, s\rangle \Rightarrow \langle S_1'\ par\ S_2, s'\rangle}$$
$$[par^2_{sos}] \quad \frac{\langle S_1, s\rangle \Rightarrow s'}{\langle S_1\ par\ S_2, s\rangle \Rightarrow \langle S_2, s'\rangle}$$
$$[par^3_{sos}] \quad \frac{\langle S_2, s\rangle \Rightarrow \langle S_2', s'\rangle}{\langle S_1\ par\ S_2, s\rangle \Rightarrow \langle S_1\ par\ S_2', s'\rangle}$$
$$[par^4_{sos}] \quad \frac{\langle S_2, s\rangle \Rightarrow s'}{\langle S_1\ par\ S_2, s\rangle \Rightarrow \langle S_1, s'\rangle}$$

'`begin` D_V S `end`' enables blocks containing local variables,
D_V is one, or many, variable declaration and is defined as $D_V ::=\texttt{begin}\ x := a; D_V|\varepsilon$. Where ε is an empty declaration.
S is then a statement using the variables defined in D_V.
The natural semantics of '`begin` D_V S `end`' is defined as

$$[block_{ns}] \quad \frac{\langle D_V, s\rangle \rightarrow_D s',\ \langle S, s'\rangle \rightarrow s''}{\langle begin\ D_V\ S\ end, s\rangle \rightarrow s''[DV(D_V) \mapsto S]}$$
$$[var_{ns}] \quad \frac{\langle D_V, s[x \mapsto \mathcal{A}[\![a]\!]s]\rangle \rightarrow_D s'}{\langle var\ x := a; D_V, s\rangle \rightarrow_D s'}$$
$$[none_{ns}] \quad \langle \varepsilon, s\rangle \rightarrow_D s$$

where $DV(D_V)$ is the set of free variables defined in D_V.

N.B. - There is also *Procedures* for *While*, look at this iff you have time.

0.2 Proofs

Proposition 0.1 - *Structural Induction*

i) *Base Case*
Prove that the property holds for all *basis* elements of a syntactic category.

ii) *Inductive Assumption*
Assume that the property holds for all immediate constituents of a *composite* element.

iii) *Inductive Case*
Prove that the property holds for all *composite* elements of a syntactic category.

Proposition 0.2 - *Induction on Shape of Derivation Trees*

i) *Base Case*
Prove that the property holds for all simple derivation trees by showing that it holds for the axiomata of the transition system.

ii) *Inductive Assumption*
For each rule assume the property holds for each of its premises.

iii) *Inductive Case*
Prove the property holds for the conclusion of all composite derivation trees (*i.e.* all rules), assuming inductive hypothesis.

Proposition 0.3 - *Induction on the Length of Derivation Sequences*

i) *Base Case*
Prove that the property holds for all derivation sequences of 0.

ii) *Inductive Assumption*
Assume the property holds for all derivation sequences of length k.

iii) *Inductive Case*
Prove the property holds for a derivation sequence of length $k+1$.

Proposition 0.4 - *Induction on the Length of Computation Sequences*

i) *Base Case*
Prove the property holds for all computation sequences of length 0.

ii) *Inductive Assumption*
Assume that the property holds for all computation sequences of length at most k.

iii) *Inductive Case*
Show that the property holds for all computation sequences of length $k+1$.

Proposition 0.5 - *Induction on the Shape of Inference Trees*

i) *Base Case*
Prove the property holds for all the simple inference trees by showing that it holds for the *axioms* of the inference system.

ii) *Inductive Assumption*
For each rule assume that the property holds for its premises

iii) *Inductive Case*
Prove the conditions of the rule are satisfied and then prove that it holds for the conclusion of the rule.

0.3 Definitions

Definition 0.1 - *Chain-Complete Partial Order*
A *Partially Ordered Set* $(D, \sqsubseteq)$ is a *Chain-Complete Partially Ordered Set* if the least upper bound, $\sqcup X$, exists $\forall$ chains $X \subseteq D$.

Definition 0.2 - *Complete Lattice*
A *Partially Ordered Set* $(D, \sqsubseteq)$ is a *Complete Lattice* if the least upper bound $\sqcup X$ exists $\forall\ X \subseteq D$ (*i.e.* All subsets, not just all chains).

Definition 0.3 - *Completeness*
If an *Inference System* is *Complete* then, if some partial correctness property does hold according to the semantics then we can find a proof for the property using the *Inference System*.
N.B. - This can be expressed as $\vdash\ \{P\}\ S\ \{Q\} \implies \vDash\ \{P\}\ S\ \{Q\}$.

Definition 0.4 - *Continuous Function*
Let $(D, \sqsubseteq)$ & $(D', \sqsubseteq')$ be chain-complete partially ordered sets and $f : D \to D'$ be a monotone function.
f is a *Continuous Function* iff $\sqcup'\{f(d)|d \in X\} = f(\sqcup X)\ \forall\ X \subset D$.

Definition 0.5 - *Fixpoints*
Consider a function $f : X \to X$.
The *Fixpoints* of a f $(fix(f))$ are the elements $x \in X$ st $f(x) = x$.
The *Least Fixpoint* of f $(lfp(f))$ is the smallest value which is also a fixpoint
i.e. $lfp(f) = x$ iff $x \in fix(f)$ & $x \leq y\ \forall\ y \in fix(f)$.

Definition 0.6 - *Functional*
Functionals are fixpoints of a function between state transformers, f : $(State \hookrightarrow State) \to$

$(State \hookrightarrow State)$.

Definition 0.7 - *Isomorphic*
Functions are said to be *Isomorphic* if there is another function such that when the two are composed (in either order) it is the same as the identity function.
Datatypes are said to be isomorphic if two functions can be defined between the two of them (in both directions) such that when the two are composed (in either order) it is the same as the identity function.
N.B. - Suppose A & B are isomorphic this is denoted as $A \cong B$.

Definition 0.8 - *Monotone Function*
Let $(D, \sqsubseteq)$ & $(D', \sqsubseteq')$ be chain-complete partially ordered sets and $f : D \to D'$.
f is *Monotone Function* iff $d_1 \sqsubseteq d_2 \implies f(d_1) \sqsubseteq' f(d_2)\ \forall\ d_1, d_2 \in D$.

Definition 0.9 - *Partial Function*
A *Partial Function* does not have a defined value for all possible input values (*i.e.* not a *Total Function*).

$$f : X \hookrightarrow Y \text{ where } \exists\ X' \subset X\ st\ \forall\ x \in X'\ \exists\ y \in Y\ st\ f(x) = y$$

N.B. - $\hookrightarrow$ in a type definition denotes a *Partial Function*.

Definition 0.10 - *Partially Ordered Set*
A *Partial Order Set* is a pairing of a set & a partial order over that set.

Definition 0.11 - *Predicate*
A *Predicate* is a function from a Program state to a boolean value.
These are used for defining properties about a program state.

Definition 0.12 - *Relation*
A *Relation* is a subset of the Cartesian product of two sets.

Definition 0.13 - *Soundness*
If an *Inference System* is *Sound* then, if some partial correctness property can be proved using the inference system, then the property does indeed hold according to semantics.
N.B. - This can be expressed as $\vdash\ \{P\}\ S\ \{Q\} \implies \vDash\ \{P\}\ S\ \{Q\}$.

Definition 0.14 - *Strong Partial Order*
A *Strong Partial Order* is a relation (consider $<$) which is

- Irreflexive - $\nexists\ x\ st\ x < x$;
- Transitive - If $x < y\ \& y < z \implies x < z\ \forall\ x, y, z$; and,

⇒ Anti-symmetric - If $x < y$ & $y < x$, suppose $x \neq y$ then $x < x$ by transitivity but this contradicts irreflexivity $\implies x = y$.

Definition 0.15 - *Total Function*
A *Total Function* has a defined value for all possible input values.

$$f : X \to Y \text{ where } \forall\ x \in X\ \exists\ y \in Y\ st\ f(x) = y$$

N.B. - $\to$ in a type definition denotes a *Total Function*.

Definition 0.16 - *Total Partial Order*
A *Total Partial Order* is a relation (consider $\leq$) which is

- Connex - $x \leq y$ or $y \leq x \ \forall \ x$;

$\Rightarrow$ Reflexive - Setting $x = y$ in connex we get $x \leq \ \forall \ x$;

- Transitive - If $x \leq y$ &$y \leq z \implies x \leq z \ \forall \ x, y, z$; and,

- Anti-symmetric - If $x \leq y$ & $y \leq x \implies x = y \ \forall \ x, y$.

N.B. - Total Partial Orders are also called chains.

Definition 0.17 - *Upper Bound of a Partially Ordered Set*
Let $X \subseteq D$. An element $d \in D$ is called an *Upper Bound* of X iff $x \sqsubseteq d \ \forall \ x \in X$.
The *Least Upper Bound* of a partially ordered set, $\sqcup X$, is the element $d \in D$ where $d \sqsubseteq d' \ \forall$ upper bounded d' of X.

Definition 0.18 - *Weak Partial Order*
A *Weak Partial Order* is a relation (consider $\leq$) which is

- Reflexive - $x \leq x \ \forall \ x$;

- Transitive - If $x \leq y$ &$y \leq z \implies x \leq z \ \forall \ x, y, z$; and,

- Anti-symmetric - If $x \leq y$ & $y \leq x \implies x = y \ \forall \ x, y$.

N.B. - $\subseteq$ is a weak partial order.

0.4 Notation

Notation 0.1 - FIX
FIX denotes the least fixpoint of a functional wrt $\subseteq$.

Notation 0.2 - $\mathcal{F}[\![\cdot]\!]$
We use the notation $\mathcal{F}[\![\cdot]\!] : X \to Y$, where $\cdot$ is a syntactic expression to denote a mapping from a syntactic category, X, to a semantic class Y. *i.e.* A map from syntax to meaning.
N.B. - Everything outside $[\![\cdot]\!]$ is semantics, everything inside $[\![\cdot]\!]$ is just syntax & has no meaning.
N.B. - $\mathcal{F}$ can be changed for any letter, to denote different functions.

Notation 0.3 - *Least Upper Bound of Partially Ordered Set*
$\sqcup X$ denotes the least upper bound of a partially ordered set X.

Notation 0.4 - *Number of Execution Steps*
We write $\gamma \Rightarrow^i \gamma'$ to denote after i steps of execution of γ we have γ'.
We write $\gamma \Rightarrow^* \gamma'$ to denote that $\exists \ n \in \mathbb{N}$ st $\gamma \Rightarrow^n \gamma'$.
N.B. - The same notation is used for all *Transition Relations*.

Notation 0.5 - *Program State*
Often a *Program State* is denoted as $\{(x_1, n_1), \dots, (x_m, n_m)\}$ where x_i is mapped to have value n_i.
Alternatively a subscript notation is used $s_{x_1 = n_1, \dots, x_m = n_m}$.
To apply a *State* s to a variable x we write $s \ x$.

Notation 0.6 - *Provability,* $\vdash$
We write $\vdash \ \{P\} \ S \ \{Q\}$ to denote that the assertion $\{P\} \ S \ \{Q\}$ is provable, by an inference tree.

Notation 0.7 - *Trans,* $\sqsubseteq$
$\sqsubseteq$ is an ordering of partial functions st if $f \sqsubseteq g$ then $f\ s = s' \implies g\ s = s'\ \forall\ s, s'$.

Notation 0.8 - *Validity,* $\vDash$
We write $\vDash\ \{P\}\ S\ \{Q\}$ if $\forall\ s\ P\ s = tt$ and $\langle S, s\rangle \to s'$ then $Q\ s' = tt$.