

Language Engineering - Notes

Dom Hutchinson

April 11, 2019

Contents

1	Programming Languages	3
1.1	Domain Specific Languages, DSL	3
1.2	Compilers	4
1.3	Case Study : Circuit Language	5
2	The Expression Problem	7
2.1	Peano Numbers	11
2.2	Composing Languages	12
3	Parsers	13
3.1	Backus-Naur Form	13
3.2	Paull's Modified Algorithm	14
3.3	Parsers	14
3.4	Alternatives	17
3.5	Monadic Parsing	18
3.6	Long Example	19
3.7	Chain for Left-Recursions	20
4	The Free Monad	21
4.1	Failure	24
4.2	Substitution	25
4.3	Non-Determinism	25
4.4	Alteration	27
4.5	State	28
4.6	Diagram of Operations	28
5	Semantics of Programming Languages	30
5.1	Properties of the Semantics	33
6	Operational Semantics	34
6.1	Natural Operational Semantics	34
6.2	Structural Operational Semantics	36
6.2.1	Configurations	36
6.2.2	Semantic Definitions	37
6.3	Provably Correct Implementation	39
6.3.1	The Abstract Machine	39
6.3.2	Correctness of Implementation	42
7	Denotational Semantics	43
7.1	Partial Orders & Fixpoints	43

8	Denotational Semantics	47
9	Static Program Analysis	49
10	Axiomatic Semantics	49
0	Reference	52
0.1	The While Language	52
0.2	Definitions	53

1 Programming Languages

Remark 1.1 - Components of Programming Languages

Programming languages have three components:

1. *Syntax*, shape & lexicon.
2. *Semantics*, meaning.
3. *Pragmatics*, purpose.

Definition 1.1 - General Purpose Programming Languages, GPL

A general purpose programming language is Turing Complete and has its own syntax & support, including compilers, ides & documentation.

Remark 1.2 - Denotation brackets, $\llbracket \cdot \rrbracket$

Denotation brackets are used to indicated the sematic evlation of a function.

Example 1.1 - Denotation Brackets

Consider `"3+5"`. This is a string, but we recognise it as an integer.

This can be indicated by $\llbracket 3 + 5 \rrbracket :: Int$.

1.1 Domain Specific Languages, DSL

Definition 1.2 - Turing Complete

A language is said to be turing complete if it can complete every possible task, within finite time.

Definition 1.3 - Domain Specific Language, DSL

A DSL is a language which has been produced with a specific purpose in mind. They are generally not Turing Complete.

Remark 1.3 - Support of DSLs

Some DSLs have a fully fledged support system, including parsers, IDEs & documentation, and are called *Standalone DSLs*.

Others will make use of the support given to other languages, these are called *embedded DSLs*. This reduces developement requirements for the DSL, but limits the flexibilty of the project.

Definition 1.4 - Embedded Domain Specific Languages, EDSL

An EDSL is is a DSL which uses the support of another language. The language it is embedded within is known as the *host language*.

EDSLs fall, generally, into two categories: *Deep Embedding*; &, *Shallow Embedding*.

Remark 1.4 - DSLs & Language Family

Standalone DSLs are common for object-orientated languages, while DSLs in functional lan- guages are more often EDSLs. This is generally believed to be due to core features of functional programming, algebraic datatypes & higher-order functions, which make producing EDSLs eas- ier.

Definition 1.5 - Deep Embedding

EDSLs which are deep embedded will have their *syntax* be concrete datatypes & *semantics* given be evaluation.

Example 1.2 - Deep Embedding

Consider a language for performing addition.

We want this DSL to subscribe the following semantic $\llbracket 3 + 5 \rrbracket :: \text{Int}$.

And then evaluate it down to $\llbracket 3 + 5 \rrbracket = \llbracket 3 \rrbracket + \llbracket 5 \rrbracket = 3 + 5 = 8$.

This can be done by deep embedding it into Haskell:

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

Now we have *concrete syntax* for "3+5"

```
Add (Var 3)(Var 5) :: Expr
```

The *semantics* can be given be

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

With these definition $\llbracket 3 + 5 \rrbracket \equiv \text{eval } (\text{Add } (\text{Var } 3)(\text{Var } 5))$.

Definition 1.6 - Shallow Embedding

EDSLs which are shallow embedded will borrow their *syntax* from the host & *semantics* is directly given.

Example 1.3 - Shallow Embedding

Implement the same language as in the deep bedding example.

Since this is shallow embedding we only use existing Haskell functions.

```
type Expr = Int

var :: Int → Expr
var n = n

add :: Expr → Expr → Expr
add x y = x + y
```

With these definitions $\llbracket 3 + 5 \rrbracket \equiv \text{add } (\text{var } 3)(\text{var } 5)$.

Definition 1.7 - Dependent Semantics

When a defined semantic depends on the result produced by another semantic, it is referred to as a *dependent semantics*.

Remark 1.5 - Implementing Dependent Semantics

Implement in deep embeddings is easy as you just call the function relating to the required semantic.

In shallow embeddings you have to pass around a tuple which holds all the properties which you are evaluating.

1.2 Compilers

Definition 1.8 - Compiler

Software which converts a source language to a target language using intermediate representation, IR.

$$\text{Source} \xrightarrow{\text{compile}} \text{IR} \xrightarrow{\text{execute}} \text{Target}$$

Definition 1.9 - Interpreter

Software which converts a source language to a target language **without** using intermediate representation.

$$Source \xrightarrow{\text{evaluate}} Target$$

Example 1.4 - Language Compilation

JavaScript is, generally, evaluated straight to its target language using an interpreter.

c is compiled to binary/assembly which is then executed in the terminal.

Haskell is compiled to binary/assembly which is then executed in the terminal when using *ghc*, but is evaluated when using **ghci**.

Definition 1.10 - Parser

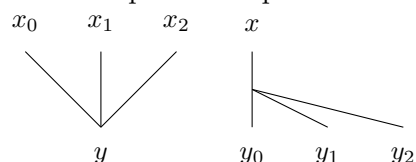
Takes a string & produces the source from it. This source can be considered as the intermediate representation in some cases.

1.3 Case Study : Circuit Language

Reference "*Folding Domain-Specific Languages: Deep and Shallow Embeddings*. Gibbons & Wu".

Definition 1.11 - Circuit

A way of displaying the flow of information. At each node connection the edges flowing in are used as inputs for a process defined for the node, this value then flows down it.



$$y = x_0 \cdot x_1 \cdot x_2 \quad x = y_0 = y_1 = y_2$$

Definition 1.12 - Circuit Language

A DSL for describing circuits.

Remark 1.6 - Operations of Circuit Language

Circuit language has several operations: **HIDDEN NOTE IN FILE.**

Remark 1.7 - Operations Descriptions

identity i - Produces a graph with *i* columns, and no connections.

beside c1 c2 - Places *c1* to the right of *c2* horizontally next to each other, adds no connections.

above c1 c2 - Places *c1* above *c2*, and connects the columns vertically.

fan i - Produces a graph with *i* columns and connects the top of the first column to the bottom of the rest.

stretch i[] c - Takes the j^{th} column of *c* and increases it to $i[j]$ columns. For any columns that were connected, the last column of their new section is connected to the last column of the new section of the target.

Example 1.5 - Deep Embedding of Circuit Language

The implementation process is:

1. Define a datastructure with abstract syntax, maybe recursive.
2. Define a semantics with an evaluation function.

Data Structure

```

data Circuit = Identity Int
             | Beside Circuit Circuit
             | Above Circuit Circuit
             | Fan Int
             | Stretch [Int] Circuit

```

Semantics

The width of a circuit is the number of vertical lines in a circuit.

Define a function to find the width of a circuit.

```

type Width = Int

width :: Circuit → Width
width (Identity n)    = n
width (Beside c1 c2)  = width c1 + width c2
width (Above c1 c2)   = width c1
width (Fan n)         = n
width (Stretch ns c) = sum ns

```

The height of a circuit is the number of layers of a circuit.

Define a function to find the height of a circuit.

```

type Height = Int

height :: Circuit → Height
height (Identity n)    = 1
height (Beside c1 c2)  = 1
height (Above c1 c2)   = height c1 + height c2
height (Fan n)         = 1
height (Stretch ns c) = 1

```

Dependent Semantics

A circuit is well connected if each layer has the same width.

A function to check this can be implemented as a dependent semantics, using the *width* function.

```

type Connected = Bool

connected :: Circuit → Connected
connected (Identity n)    = True
connected (Beside c1 c2)  = connected c1 ∧ connected c2
connected (Above c1 c2)   = connected c1 ∧ connected c2
                           ∧ (width c1 ≡ width c2)
connected (Fan n)         = True
connected (Stretch ns c) = connected c ∧ (width c ≡ length ns)
                           ∧ !(elem 0 ns)

```

Remark 1.8 - Extending Deep Embedding

To extend this deep embedding to include more semantics would be easy, just define more functions.

To include new properties in the data type we would need to rewrite all the functions using the data type to work with this new property.

Example 1.6 - Shallow Embedding of Circuit Language

Implement a semantics to find the width of a circuit.

```

type Width = Int

```

```

type Circuit = Width

identity :: Int → Circuit
identity n = n

beside :: Circuit → Circuit → Circuit
beside c1 c2 = c1+c2

above :: Circuit → Circuit → Circuit
above c1 c2 = c1

fan :: Int → Circuit
fan n = n

stretch :: [Int] → Circuit → Circuit
stretch ns c = sum ns

```

Remark 1.9 - Extending Shallow Embedding

To extend this shallow embedding to include more semantics would be difficult as we would have to rewrite all the functors.

However, it would be easy to add new properties to the datastructure as we would just need to implement a new function for this property.

2 The Expression Problem

Definition 2.1 - The Expression Problem

The *expression problem* solves the problem

Is it possible to extend the syntax & semantics of a language in a modular way?

Definition 2.2 - Type Constructor

A function which takes a type and returns a type.

Definition 2.3 - Functor

A functor is a group of data structures which can be mapped over by the same function, F .

Functors must obey two rules

- i) Identity, $F \text{ Identity} = \text{Identity}$.
- ii) Composition, $F(g \cdot f) = F(g) \cdot F(f)$.

Definition 2.4 - fmap

$fmap$ is the generalised map function for all functors, which all functors implement.

The signature of $fmap$ is

```
fmap :: (a → b) → f a → f b
```

f is a type constructor So $fmap$ takes a function & a data structure of elements, applies the function to all elements of the data structure and returns the processed datastructure.

Remark 2.1 - Implementations of fmap

$fmap$ is implemented in different instances of *Functor*.

Here is an implementation of $fmap$ of arrays

```
instance Functor [] where
  fmap f [ ] = [ ]
  fmap (x:xs) = (f x) : (fmap f xs)
```

Definition 2.5 - Syntatic Sugar List Definition

The standard definition for lists in Haskell uses syntatic sugar

```
data [a] = [ ]
         | α : [a]
```

Definition 2.6 - Syntatic-Sugarless List Definition

this definition uses syntatic sugar. Removing this exposes its generic structure.

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

Definition 2.7 - Recursiveless List Definition

We remove recursion from the *List* datatype & use *k* to mark a *kontinuation* point

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

This is a functor expression.

Example 2.1 - Representing lists as ListFs

```
[5]    ≡ in (ConsF 5 (in EmptyF))
[6,7]  ≡ in (ConsF 6 (in (ConsF 7 (in EmptyF))))
```

Definition 2.8 - Mutual Recursion

If two functions or datatypes are defined in terms of each other, they are said to be *mutually recursive*.

Definition 2.9 - Fix

The datatype *Fix* provides a fixed point of data. It is defined as

```
data Fix f = in (f (Fix f))
```

where *f* is a Functor.

Fix allows us to generalise all non-mutually recursive datatypes.

N.B. - When considering *Fix* consider *Maybe* without a paramter.

Definition 2.10 - inop

inop is the opposite of *in*.

It is defined as

```
inop :: Fix f → f (Fix f)
inop (in x) = x
```

Definition 2.11 - Isomorphic

Two datastructures, *a* & *b*, are isomorphic if there exists two functions, *to* & *from*, st

```
to    :: a → b
from  :: b → a
```

```
(to . from id) = id
(from . to id) = id
```

Definition 2.12 - ListF Functor instance

A recursiveless list structure, *ListF*, can be implemented by


```

instance Functor (ListF a) where
  fmap :: (x → y) → ListF a x → ListF a y
  fmap f EmptyF      = EmptyF
  fmap f (Consf a x) = Cons a (fx)

```

Definition 2.13 - Catamorphism

Provides a generalisation of Folds.

A catamorphism arises from

$$\begin{array}{ccc}
 & \text{fmap cata alg} & \\
 f(\text{Fix } f) & \xrightarrow{\quad} & f \ b \\
 \text{In } \downarrow & \swarrow \searrow & \downarrow \\
 & \text{inop} & \text{algebra, alg} \\
 \downarrow & & \downarrow \\
 \swarrow \searrow & \text{cata alg} & \swarrow \searrow \\
 \text{Fix } f & \xrightarrow{\quad} & b
 \end{array}$$

Definition 2.14 - Catamorphism Implicit Definition

By following the arrows of the diagram we can define *cata*.

```

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)

```

N.B. - $(f \ b \rightarrow b)$ is called the algebra.

Proposition 2.1 - $\text{Fix}(\text{ListF } a)$ & $\text{List } a$ are isomorphic

With this being true, we can write $\text{Fix}(\text{ListF } a)$ instead of $\text{List } a$.

```

fromList :: List a → Fix (ListF a)
fromList Empty      = in (EmptyF)
fromList (Cons x xs) = in (Cons F x (fromList xs))

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

```

Alternatively

```

toList' :: Fix (ListF a) → [a]
toList' = cata alg
  where
    alg :: ListF a [a] → [a]
    alg EmptyF      = []
    alg (ConsF x xs) = x : xs

```

N.B. - k has been dropped from ListF declaration as it is provided by *Fix*.

Proposition 2.2 - Implementations of *cata* - List Length

Here we define a funtion which returns the length of a $\text{Fix}(\text{ListF } a)$.

```

length :: Fix (ListF a) → Int
length = cata alg
  where

```

```

alg :: ListF a Int → Int
alg Empty F      = 0
alg (ConsF x y) = 1 + y

```

N.B. - y is the length of list calculated so far.

Example 2.2 - *Lazy Evaluation of length*

Here we shall perform a lazy evaluation of *length* of $[7, 9]$.

```

[7,9] ≡ in (ConsF 7 (in (ConsF 9 (in EmptyF))))

length (in (ConsF 7 (in (ConsF 9 (in EmptyF)))))
{length}
cata alg (in (ConsF 7 (in (ConsF 9 (in EmptyF)))))
{cata}
alg (fmap (cata alg) (ConsF 7 (in (ConsF 9 (in (ConsF 9 (in EmptyF)))))))
{fmap}
alg (ConsF 7 (cata alg) (in (ConsF 9 (in EmptyF))))
{alg}
1 + cata alg (in (ConsF 9 (in EmptyF)))
{cata}
1 + alg (fmap (cata alg) (ConsF 9 (in EmptyF)))
{fmap}
1 + alg (ConsF 9 (cata alg) (in EmptyF))
{alg}
1 + 1 + cata alg (in EmptyF)
{cata}
1 + 1 + alg (fmap (cata alg) EmptyF)
{fmap}
1 + 1 + alg EmptyF
{alg}
1 + 1 + 0
{+}
2

```

Proposition 2.3 - *Implementations of cata - Sum of Integer List*

Here we define a function which returns the sum of a *Fix*(*ListF* *Int*).

```

sum :: Fix (ListF Int) → Int
sum = cata alg
  where
    alg :: ListF Int Int → Int
    alg EmptyF      = 0
    alg (ConsF x y) = x + y

```

N.B. - y is the sum so far

Example 2.3 - *Lazy Evaluation of sum*

Here we shall perform a lazy evaluation of *sum* of $[7, 9]$.

```

[7,9] ≡ in (ConsF 7 (in ConsF 9 (in EmptyF)))

sum (in (ConsF 7 (in (ConsF 9 (in EmptyF)))))
{sum}
cata alg (in (ConsF 7 (in (ConsF 9 (in EmptyF)))))
{cata}
alg (fmap (cata alg) (in (ConsF 7 (in (ConsF 9 (in EmptyF))))))

```

```

{fmap}
alg (ConsF 7 (cata alg) (in (ConsF 9 (in EmptyF))))
{alg}
7 + cata alg (in (ConsF 9 (in EmptyF)))
{cata}
7 + alg (fmap (cata alg) (ConsF 9 (in EmptyF)))
{fmap}
7 + alg (ConsF 9 (cata alg) (in EmptyF))
{alg}
7 + 9 + cata alg (in EmptyF)
{fmap}
7 + 9 + alg (fmap (cata alg) EmptyF)
{alg}
7 + 9 + 0
{+}
16

```

2.1 Peano Numbers

Definition 2.15 - Peano Number

A *peano number* is either zero or the successor of another peano number. They have the generic form of

```

data Peano = Z          — zero
           | S Peano    — successor

```

Example 2.4 - Peano number

The number 3 can be represented as a peano number by

$$3 = S(S(S Z))$$

Proposition 2.4 - Peano Number Functor

The functor signature for peano numbers is

```

data PeanoF = ZF
           | SF k

```

The functor instance for peano numbers is

```

instance Functor PeanoF where
  fmap :: (a → b) → PeanoF a → PeanoF b
  fmap f ZF      = ZF
  fmap f (SF x)  = SF (f x)

```

Proposition 2.5 - Implementations of cata - Peano Number to Integer

This function converts peano numbers to integers

```

toInt :: Fix PeanoF → Int
toInt = cata alg
  where
    alg :: PeanoF Int → Int
    alg ZF      = 0
    alg (SF x)  = 1 + x

```

Proposition 2.6 - Implementations of cata - Double Value of Peano Number

This function doubles the value of a peano number by double the number of *SF* values.

```

double :: Fix PeanoF → Fix PeanoF
double = cata alg
  where
    alg :: PeanoF → Fix PeanoF
    alg ZF      = in ZF
    alg (SF x)  = in (SF in (SF x))

```

2.2 Composing Languages

Definition 2.16 - Coproduct of Functors

The *coproduct of functors* is defined as

```

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

```

So given two functors f & g , it produces a functor $(f :+: g)$ which executes a different function depending on the type of the input a .

The functor instance is

```

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

```

N.B. - $x :: f\ a$ & $y :: g\ a$.

Definition 2.17 - Junction of Algebras

To reduce a *Fix* expression on a *coproduct of functors* to simple $a \rightarrow b$ we use a *junction of algebras*. Defined as

```

(∇) :: (f a → a) → (g a → a) → (f :+: g) a → a
(flag ∇ galg)(L x) = falg x
(flag ∇ galg)(R x) = galg x

```

Example 2.5 - Coproduct of Functors

We previously defined a language for addition.

```

data Expr = Val Int
          | Add Expr Expr

```

Which can be expressed without recursion as

```

data ExprF = ValF Int
          | AddF k k

```

So $Fix\ ExprF = Expr$.

A functor signature for multiplication is

```

MulF' :: k → k → MulF k
MulF k = MulF' k k

```

This can be instantiated with

```

instance Functor MulF where
  fmap f (MulF x y) = Mul (f x) (f y)

```

By defining the algebras

```

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 use the coproduct of functors to create a single evaluation for both addition & multiplication.

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

3 Parsers

3.1 Backus-Naur Form

Definition 3.1 - *Backus-Naur Form, BNF*

Backus-Naur Form is a language used to express the shape of grammar. It was invented in 1958 for express of Algol.

Remark 3.1 - *Syntax of Backus-Naur Form*

A *Backus-Naur Form* statement is made up of

- ε , empty string.
- $\langle a \rangle$, non-terminal.
- " a ", terminal.
- $a|b$, choice between a or b .

Remark 3.2 - *Additional Syntax of Backus-Naur Form*

Backus-Naur Form is usually extended to include the following, although they can be achieved using the previous features.

- $[a]$, optional a .
- (a) , grouping a .
- a^* , ≥ 0 repetitions a .
- a^+ , ≥ 1 repetitions a .

Example 3.1 - *Expressing Numbers in BNF*

The following represents a single digit

```
<digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

This can be used to approximate numbers

```
<num> ::= <digit>
        | <digit> <num>
```

Example 3.2 - *Expressing Addition in BNF*

In Haskell we define addition as

```
data Expr = Val Num
          | Add Num Expr
```

In BNF it can be expressed as

$$\begin{aligned} \langle \text{expr} \rangle &::= \langle \text{num} \rangle \\ &\quad | \langle \text{num} \rangle \text{ "+" } \langle \text{expr} \rangle \end{aligned}$$
Remark 3.3 - Ambiguous Grammars

Some grammar definition are not precise enough, as they can be interpreted in multiple ways.

Example 3.3 - Ambiguous Grammars

Consider the following definition for addition, in BNF

$$\begin{aligned} \langle \text{expr} \rangle &::= \langle \text{num} \rangle \\ &\quad | \langle \text{expr} \rangle \text{ "+" } \langle \text{expr} \rangle \end{aligned}$$

This is ambiguous as, should it be given $1 + 2 + 3$ it doesn't define whether to evaluate it as $(1 + 2) + 3$ or $1 + (2 + 3)$.

N.B. - The issue with this definition occurs in *recursive-descent parses*.

3.2 Paull's Modified Algorithm**Definition 3.2 - Paull's Modified Algorithm**

Paull's Modified Algorithm is used to remove recursion from grammar definitions.

Given

$$A ::= A\alpha_1 \mid \dots \mid A\alpha_n \mid \beta_1 \mid \dots \mid \beta_m$$

where A is a non-terminal and α_i, β_j are *BNF* expressions.

Applying *Paull's Modified Algorithm* refactors this to

$$\begin{aligned} A &::= \beta_1 A' \mid \dots \mid \beta_m A' \\ A' &::= \alpha_1 A' \mid \dots \mid \alpha_n A' \mid \varepsilon \end{aligned}$$
Example 3.4 - Paull's Modified Algorithm

Given the expression

$$\begin{aligned} \langle \text{expr} \rangle &::= \langle \text{num} \rangle \\ &\quad | \langle \text{expr} \rangle \text{ "+" } \langle \text{expr} \rangle \end{aligned}$$

this is refactored to

$$\begin{aligned} \langle \text{expr} \rangle &::= \langle \text{num} \rangle \langle \text{expr}' \rangle \\ \langle \text{expr}' \rangle &::= \text{ "+" } \langle \text{expr} \rangle \langle \text{expr}' \rangle \mid \varepsilon \end{aligned}$$
3.3 Parsers**Definition 3.3 - Parser**

A parser is a function that takes in a list of characters (string) & returns an item that has been parsed, along with the unconsumed string.

The haskell data type for this is

$$\text{newtype Parses } a = \text{Parse } (\text{String} \rightarrow [(a, \text{String})])$$
Proposition 3.1 - Parse

We can use a parser by defining the function *parse*.

$$\begin{aligned} \text{parse} &:: \text{Parser } a \rightarrow \text{String} \rightarrow [(a, \text{String})] \\ \text{parse } (\text{Parser } px) \text{ s} &= px \text{ s} \end{aligned}$$
Proposition 3.2 - Fail Parser

This parser always returns an empty list, showing the parse has failed.

$$\begin{aligned} \text{fail} &:: \text{Parser } a \\ \text{fail} &= \text{Parser } (\lambda \text{ ts} \rightarrow []) \end{aligned}$$

When implemented we get

```
parse fail "Hello" = [ ]
```

Proposition 3.3 - Item Parser

This parser returns a single character when passed a string.

```
item :: Parser Char
item = Parser (\ ts → case ts of
    [] → []
    (t:ts') → [(t, ts')])
```

When implemented we get

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

Proposition 3.4 - Look Parser

This parser is used to look at the input stream without consuming anything.

```
look :: Parser String
look = Parser (\ ts → [(ts, ts)])
```

When implemented we get

```
parse look "Hello" = [("Hello", "Hello")]
```

Remark 3.4 - Transforming Parsers

Sometimes we want to transform our parses so they produce values of a different type. this is achieved by given a functor instance for parses.

```
instance Functor Parser where
    fmap f (Parser ps) = Parser (\ ts → [(f x, ts') | (x, ts') ← ts])
```

This can be visualised as

ts	
x	ts'
f x	ts'

Definition 3.4 - Combinations

Combinations are used to combine parsers, they have different definitions which can be utilised for a desired effect.

```
(<$>) :: (a → b) → Parser a → Parser b
f <$> px = fmap f px
```

There are variations on this, such as

```
(<$) :: a → Parser b → Parser a
x <$ py = fmap (cost x) py
```

Remark 3.5 - Space is a Combinator

Consider $f x$, the space between f and x can be considered as an operation which takes f & x as inputs and then applies f to x .

Proposition 3.5 - Skip Parse

This function parses an input but outputs nothing useful.

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

Definition 3.5 - Applicative Class

This class defines the functions *pure* and $(< * >)$.

```

class Functor f => Applicative f where
  pure  :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b

```

Proposition 3.6 - Parser Applicative Class Instance

The Applicative class can be instantiated for Parsers as

```

instance Applicative Parser
  where
    pure x = Parse (\ts -> [(x,ts)])
    (Parser pf) <*> (Parse px) = Parser (\ts ->
      [ (f x, ts'')
      | (f,ts') <- pf ts
      , (x,ts'') <- px ts' ])

```

The defined operation first parses a function, then a value and then applies the function to this value.

Proposition 3.7 - Variations of ap operator
Inverse Ap

Say we wanted to parse a value, then parse a function and then apply the function to the value we use the following.

```

(<*>) :: Parser a -> Parser (a -> b) -> Parser b
(Parser px) <*> (Parser pf) = Parse (\ts ->
  [ (f x, ts'')
  | (x,ts') <- px ts
  , (f,ts'') <- pf ts' ])

```

Left ap

If we want to only use the Parser to the left of the operator we use the following.

```

(<*) Parser a -> Parse b -> Parse a

```

Right ap

If we want to only use the Parser to the right of the operator we use the following.

```

(*>) Parser a -> Parse b -> Parse b

```

Remark 3.6 - Implementation of Applicative Parser

When applied we get $\text{parse}(\text{pure } 5) \text{ "Hello"} = [(5, \text{"Hello"})]$.

Proposition 3.8 - Apply Function of Parse Output

Consider a parse which results in

ts				
(a→b→x)	x::a	y::b	z::c	ts'

We want to apply a function to the different outputs of the parse. To make this function which does this we can use $< * >$ as

```

f <*> px <*> py <*> pz

```

N.B. - The f here is given to us & we didn't need to parse it.

Definition 3.6 - Monoidal Type Class

The Monoidal type class is equivalent to Applicative.


```

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

```

Proposition 3.9 - Parser Monoidal Class Instance

The parser datatype can be instanced for monoidal class by

```

instance Monoidal Parser
  where
    unit = Parser (\ ts → [(() , ts)])
    mult (Parser px) (Parser py) = Parser (\ ts → [ ((x,y), ts'')
      | (x, ts') ← px ts
      | (t, ts'') ← py ts'])

```

Here the x is parsed first and then y is parsed from the string remaining from that.

Remark 3.7 - *mult* as a Binary Operator

It is useful to make *mult* a binary operator

```

(<~>) :: Monoidal f ⇒ f a → f b → f (a,b)
px <~> py = mult px py

```

Remark 3.8 - Variations of *mult* Binary Operator

We can derive left and right variables of $< \sim >$.

```

(<~) :: Monoidal f ⇒ f a → f b → f a
px <~ py = fst <$> (px <~> py)

```

```

(~>) :: Monoidal f ⇒ f a → f b → f b
px ~> py = snd <$> (px <~> py)

```

3.4 Alternatives**Definition 3.7 - Alternative Type Class**

We can produce parsers that can deal with choices in a grammar

```

class Alternative f
  where
    empty :: f a
    (<|>) :: f a → f a → f a

```

Remark 3.9 - Parser Alternative Class Instance

The Parse datatype can be instanced for the Monoidal class by

```

instance Alternative Parser
  where
    empty = fail
    (Parser px) <|> (Parser py) = Parser (\ ts → px ts ++ py ts)

```

Remark 3.10 - Equivalents to $< | >$

```

parse (px <|> py) ts = (parse px ts) ++ (parse py ts)

```

Definition 3.8 - choice

choice is used to extend $< | >$ to accept many input parsers.

```
choice :: [Parser a] → Parser a
choice pxs = foldr (<|>) pxs
```

Definition 3.9 - Combinator - <:>

We can define a combinator that appends the result of a parse into others

```
(<:>) :: Parser a → Parser [a] → Parser [a]
px <:> pxs = (:) <$> px <*> pxs
```

N.B. - $(:) <$> px <*> pxs = ((:) <$> px) <*> pxs$.

Proposition 3.10 - Backus-Naur Form Parsers

We can define combinators that correspond to '+' & '*' from Backus-Naur Form.

i) $e+$ is written *some e*.

ii) e^* is written *many e*.

```
some :: Parser a → Parser [a]
some px = px <:> many px
many :: Parser a → Parser [a]
many px = some px <|> pure
```

some parses one px and adds it to the result of *many px*.

many px returns *some px*, if this does not exist then returns *empty*.

3.5 Monadic Parsing**Remark 3.11 - Motivation**

Sometimes we want to control the flow of a *Parser* to depend on what was parsed.

Suppose we have $px :: \text{Parser } a$ we can define a function $f :: a \rightarrow \text{Parser } b$.

The function f inspects the value x which came from px and produces a new Parser accordingly. the result should be a parser of type *Parser b*.

Definition 3.10 - Monadic Instance of Parser

```
instance Monad Parser where
  -- return :: a → Parser a
  return = pure          -- From Applicative
  -- (>>=) :: Parser a → (a → Parser b) → Parser b
  (Parser px) >>= f = Parser (\ts to
                                concat [parse (fx) ts'
                                           | (x,ts') ← px ts])
```

Proposition 3.11 - Implementing Monadic Combinator

To use the monadic combinator we combine a parser with a function.

The *satisfy* parser takes a function that is a predicate of *Chars* and returns the parse value if it satisfies the predicate.

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

Proposition 3.12 - satisfy Without Monadic Definition

satisfy can be defined without using the monadic definition.

```

satisfy :: (Char→Bool)→Parser Char
satisfy p = Parser (λ ts to case ts of
    [] →empty
    (t:ts ') →[(t,ts ')|p t])

```

Definition 3.11 - Char Parser

We can parse a single character as follows

```

char :: Char→Parser Char
char c = satisfy (c==)
OR
char c = satisfyt (λ c' →c'==c)

```

N.B. - parse (char 'x')"xyz" = [(x',"yz")].

3.6 Long Example

Consider beginning given the following definitions:

```

expr = term | expr '+' term
term = fact | term '*' fact
fact = numb | '(' expr ')'
numb = '0' | '1' | ... | '9'

```

These definitions are left recursive which is bad as the compiler can end up in a loop.

We can change this by using Paull's Algorithm.

```

expr = term expr '
expr ' = '+' term expr ' | ε

term = fact term '
term ' = '*' fact term ' | ε

```

From these we can make corresponding datatypes.

Each non-terminals becomes a datatype, whereas the alternations are constructors.

```

data Expr = ExprTerm Term Expr '
data Expr' = Expr'Add Term Expr' | ExprEps '

data Term = TermFact Fact Term '
data Term' = Term'Mul Fact Term' | TermEps '

data Fact = FactNumb Int | FactPar Expr

```

Now we have usable datastructure we can make a Parser.

These correspond to Paull's Definition.

```

expr :: Parser Expr
expr = Expr <$> term <*> expr

expr' :: Parse Expr'
expr' = Expr'Add <$ char '+' <*> term <*> expr'
      <|> Expr'Eps <$ pure()

term :: Parser Term
term = TermFact <$> fact <*> term'

```

```

term' :: Parser Term'
term' = Term'Mul <$ char '*' <*> fact <*> term'
      <|> Term'Eps <$ pure ()

fact = FactNumb <$> numb
      <|> FactPar <$ char '(' <*> expr <*> char ')',

numb :: Parser Int
numb = read <$> digits

digit :: Parser Char
digit = satisfy isDigit

digits :: Parser Int
digits = cull (some digit)

```

These can be applied to produce the following results

```
parse (numb) "12hh3124" = [(12, "hh3124")]
```

Remark 3.12 - Debugging

By enabling *derive Show* on all the previous structures will print when to terminal when required, making debuggin easier.

Remark 3.13 - When to use <>, < or >

Often it is best to write out the expression in full and then remove parentenses are required.

3.7 Chain for Left-Recursions

Remark 3.14 - Motivation

The problem with ambiguous grammars, which are left recursive, can be solved with *Paull's Algorithm*.

```

<expr> ::= <number>
        | <expr> "+" <expr>

```

However, without applying *Paull's Algorithm* we have a nice datatype

```

data Expr = Num Int
          | Add Expr Expr

```

Definition 3.12 - *chainl₁*

chainl₁ is used to parse from the original grammar into this datastructure, assuming the operation ("+") is left associative.

It's signature is

```
chainl1 :: Parser a → Parser (a → a → a) → Parser a
```

N.B. - *chainr₁* is used when the operation is right associative.

Example 3.5 - *chainl₁*

Consider making a parser for the following datatype

```

data Expr = Num Int
          | Add Expr Expr

```

chainl₁ can be used to achieve this

```

expr :: Parser Expr
expr = chainl1 number add

```

This can be expanded to create a parser for addition

```

add :: Parser (Expr → Expr → Expr)
add = Add <$ tok "+"

```

4 The Free Monad

AKA Generic Syntax or Abstract Syntax Trees.

Proposition 4.1 - Abstract Syntax Tree

When analysing the syntax of a language, we first want to establish which symbols are operations and which are variables.

We can use a tree to show the relationships between variables and operations.

The elements in diamonds are operations & those in rectangles are variables.

Remark 4.1 - Compiler

Operation symbols tell the compiler to add command to stack.

Variable symbols tell the compiler to load the value into memory and add to stack.

Example 4.1 - Abstract Syntax Trees

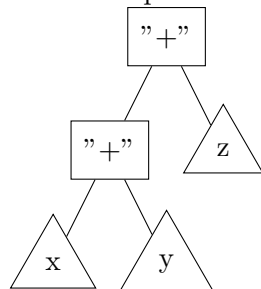
Suppose we are interested in giving a semantics to a language for addition.

The syntax could look like

$$(x + y) + z$$

Here we are not considering whether x , y & z are numbers yet.

This example can be consider as the syntax tree

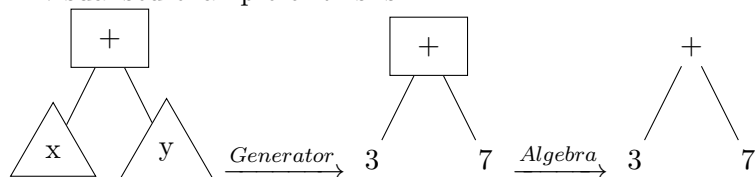


Remark 4.2 - Interpreting Free Monad Trees

We interpret free monad trees in two stages:

- i) Generator - Change variables into values;
- ii) Algebra - Evaluate the operations using these values.

A visualised example of this is



Definition 4.1 - Free Monad

The free monad $Free\ f\ a$ provides syntax trees whose nodes are shaped by *functor* f and whose

variables come from the type a .

It is defined as

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

N.B. - The Op constructor is similar to the definition for Fix .

Proposition 4.2 - Functor Instance of Free Monad

Consider $(Op\ op) :: Free\ f\ a$, here $op :: f(Free\ f\ a)$.

Using $fmap$ we want to make it of type $f(Free\ f\ b)$.

This can be done using $fmap(fmap\ f)$ since $fmap :: f(Free\ f\ a) \rightarrow f(Free\ f\ b)$ which will evaluate the outside of op and $(fmap\ f) :: Free\ f\ a \rightarrow Free\ f\ b$ which will evaluate the inside of op .

```
instance Functor f => Functor (Free f)
  where
    — fmap :: (a->b)->(Free f a)->(Free f b)
    fmap f (Var x) = Var (f x)
    fmap f (Op op) = Op (fmap (fmap f) op)
```

Example 4.2 - Free Monad - Addition

Consider the double addition example.

We can define the *shape* of $+$

```
data AddF k = AddF k k
OR data AddF k = k :+: k
```

N.B. - The tree for double addition can be expressed with the following values of type $Free\ AddF\ String$ and $Op\ (AddF\ (Op\ (AddF\ (Var\ "x")(Var\ "y")))(Var\ "z"))$.

Definition 4.2 - extract

The second stage extracts semantics by applying an algebra.

This is done by a recursive function defined as

```
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. - The second clause here is identical to *cata*.

Definition 4.3 - eval

By combining the generator & algebra stage we can produce an evaluation function

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

N.B. - $fmap\ gen$ executes the first stage & $extract\ alg$ executes the second.

Example 4.3 - Addition Language

Consider the $AddF$ functor from earlier.

First we define an algebra for the functor

```
alg :: AddF Int->Int
alg (x :+: y) = x + y
```

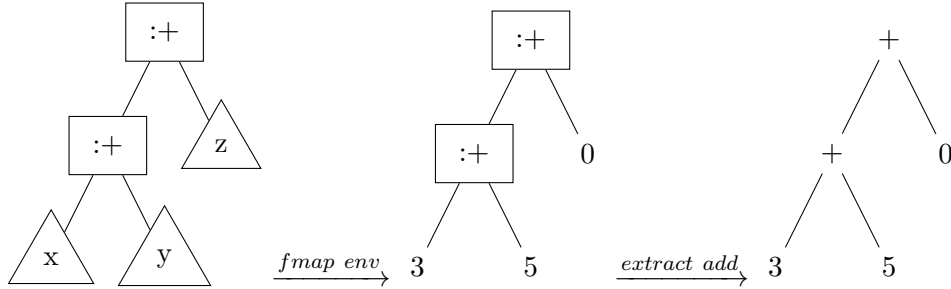
Now we define a generator from the type of our variables to integers.

```

type Var = String
env  :: Var → Int
env  "x" = 3
env  "y" = 5
env  _   = 0

```

Here is a visualised example of the evaluation process



N.B. - This is the evaluation of $eval\ add\ env$.

Example 4.4 - Variable Collector

In this example we will collect all the variables in an expression as a list.

So $"x + y + z" \rightarrow ["x", "y", "z"]$.

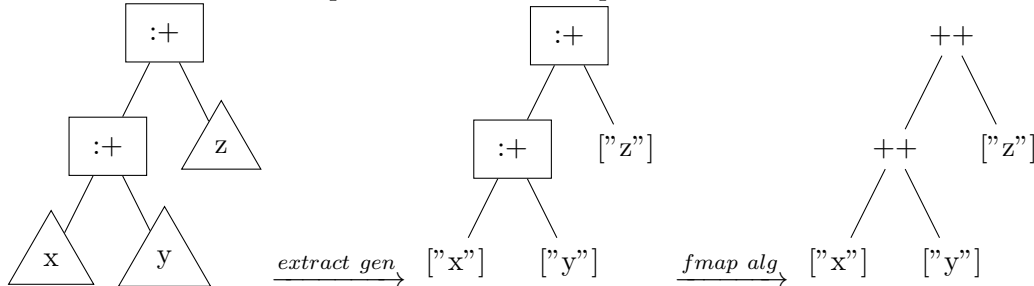
We define a function for this as

```

vars :: Free AddF Var → [Var]
vars = eval alg gen
  where
    — gen :: Var → [Var]
    — alg :: AddF [Var] → [Var]
    gen x      = [x]
    alg (xs :+: ys) = xs ++ ys

```

Here is a visualised example of the evaluation process



Proposition 4.3 - Division Operation

Suppose we want to add an new operation to our arithmetic language that performs division.

We can do this with the following datatype

```

data DivF k = DivF k k

```

If we want to provide a semantics that collects all the variables we must provide an algebra as follow

```

divVars :: DivF [Var] → [Var]
divVars (DivF xs ys) = xs ++ ys

```

If we only want to work with division we can use the following

```

evalDiv :: Free DivF Var → [Var]
evalDiv = eval alg gen
  where

```

```

gen :: Var → [Var]
gen x = [x]
alg :: DivF [Var] → [Var]
alg (DivF xs ys) = xs ++ ys

```

If we want a language with both addition and division we need to take the co-product of *AddF* and *DivF*

```

vars :: Free (AddF :+: DivF) Var → [Var]
vars = eval alg gen
where
  gen x = [x]
  alg :: (AddF :+: DivF) [Var] → [Var]
  alg (L (AddF xs ys)) = xs ++ ys
  alg (R (DivF xs ys)) = xs ++ ys

```

When performing division we cannot simply do $\frac{x}{y}$ incase $y = 0$.
Then we use the *Maybe* datatype

```

expr :: Free (AddF :+: DivF) Var → Maybe Double
expr = eval alg gen
where
  gen = env[1]
  alg (L (AddF x y)) = mAdd x y
  alg (R (DivF x y)) = mDiv x y

mAdd :: Maybe Double → Maybe Double → Maybe Double
mAdd (Just x) (Just y) = Just (x+y)
mAdd _ _ = Nothing

mDiv :: Maybe Double → Maybe Double → Maybe Double
mDiv _ (Just 0) = Nothing
mDiv (Just x) (Just y) = Just (x/y)
mDiv _ _ = Nothing

```

4.1 Failure

Definition 4.4 - *Fail Datatype*

We need a syntax for failure

```
data Fail k = Fail
```

Proposition 4.4 - *Functor instance of Fail*

The functor instance of *Fail* shows us that computations cannot follow a fail.

```

instance Functor Fail
where
  fmap f Fail = Fail

```

Proposition 4.5 - *Fail and Division*

Here we integrate *Fail* with the division operation defined in **Section 4.0**.

```

evalFail :: Free DivF Double → Free Fail Double
evalFail = eval alg gen
where
  gen :: Double → Free Fail Double

```



```

gen x=Var x
alg :: DivF (Free Fail Double) → Free Fail Double
alg (DivF (Var x) (Var 0)) = Op Fail
alg (DivF (Var x) (Var y)) = Var (x/y)
alg - = Op Fail

```

This process converts a tree to another tree, checking for possible failures.

4.2 Substitution

Remark 4.3 - Motivation

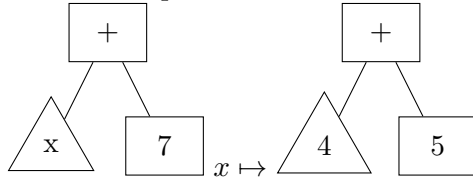
Consider $'x+7'$.

We can evaluate this into a new syntax tree where we have a notion of substitution, where we might bind $'x'$ to another expression rather than a constant $x \mapsto 4 + 5$.

Then we expect $x + 7 \mapsto (4 + 5) + 7$.

Proposition 4.6 - Tree Representation

The initial expression and substitution in **Remark 4.3** can be represented by the following trees



N.B. - Here we consider $'4'$, $'5'$ & $'7'$ to be operations.

Remark 4.4 - Notation

Consider an expression e with a variable x , where x is substituted by e' .

We denoted this by

$$e[x \mapsto e']$$

Occasionally $e[x \setminus e']$ or $e[e'/x]$ are used.

Definition 4.5 - ($>>=$)

Substitution is defined by ($>>=$)

```

(>>=)::Free f a → (a → Free f b) → Free f b
Var x >>= f = f x
Op op >>= f = Op (fmap (>>= f) op)

```

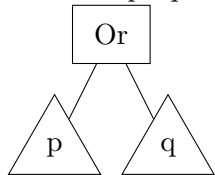
This applies the substitution, ($>>=$) f , to all lower layers of the tree.

4.3 Non-Determinism

Definition 4.6 - Non-Deterministic Computation

A *Non-Deterministic Computation* is a computation that provides the choice between two different computations.

Consider $p \sqcap q$ is the program that gives answers p or q



Remark 4.5 - Square Root

The square root is a non-deterministic computation as it produces a pair or results, one positive

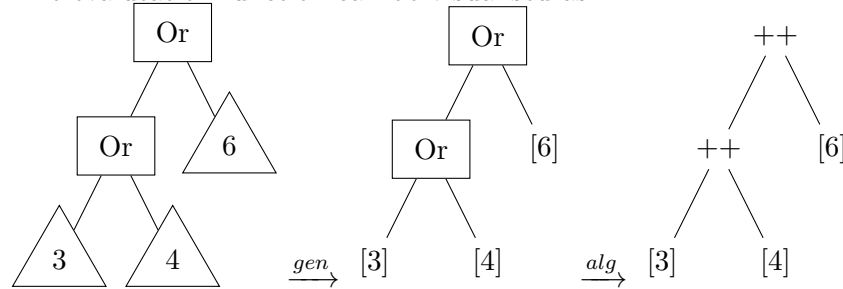
& one negative.

Example 4.5 - Non-Deterministic Computation

Suppose a computation produces the possible results 3,4 or 6.

We want to return a list containing these results.

The evaluation function can be visualised as



Proposition 4.7 - Implementation

We can express non-determinism by the syntax

```
data Or k=Or k k
```

We need this to be a function in order to use the evaluator

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

Now we can define an evaluation function

```
list :: Free Or a -> [a]
list = eval alg gen
where
  gen :: Free Or a
  gen x = [x]
  alg :: Or [a] -> [a]
  alg (Or xs ys) = xs ++ ys
```

Another interpretation of these trees is to simply return the first result.

This can be done by the following code

```
once :: Free Or a -> Maybe a
once = eval alg gen
where
  gen :: Free Or a -> Maybe a
  gen x = Just x
  alg :: Or (Maybe a) -> Maybe a
  alg (Or Nothing y) = y
  alg (Or (Just x) y) = Just x
```

Definition 4.7 - Non-Determinism Syntax

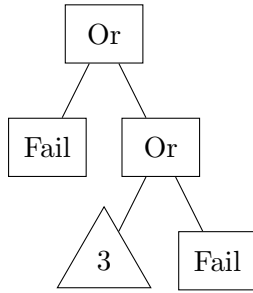
The above approach works but we want to signal if there is no solution.

We do this using Nondeterminism, which can be given by the following syntax

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

Remark 4.6 - Nondet Trees

Trees of type *Free (Nondet) a* have shape

**Definition 4.8 - Semantics for Nondet Languages**

We give semantics to nondet languages by providing a generator and an algebra.

```

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

```

N.B. - The semantics for *once* is similar.

4.4 Alteration**Remark 4.7 - Motivation**

A different approach, than the *Or*, is to model a pair of k values as a function from *Bool*.

Definition 4.9 - Alternative Datatype

We can define a data type for this as

```
data Alt k=Alt (Bool→k)
```

We pass *True* when we want the first child and *False* for the second.

Definition 4.10 - Alt Functor Definition

The *Functor* instance for *Alt*

```

instance Functor Alt
  where
    fmap f (Alt k)=Alt (f.k)

```

Definition 4.11 - Non-Determinism Datatype

We can give alternative semantics for non-determinism using *Alt*

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

Definition 4.12 - list

We can redefine *list* using the new *Nondet'* definition

```

list :: Free Nondet' a→[a]
lsit=eval alg gen
  where
    gen :: a→[a]
    gen x=[x]
    alg :: Nondet' [a]→[a]

```

```

alg (L Fail)    = []
alg (R (Alt k)) = (k True) ++ (k False)

```

N.B. - This shows that the parameter to a syntax function sometimes has the form of a function.

4.5 State

Definition 4.13 - Stable Computation

A stable computation can be modelled by having two operations *GET* & *Put*.

These can be defined as

```

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

```

Remark 4.8 -

The intuition of *State* is that *Put s k* will put the value *s* into the state, before continuing with the computation in *k*.

The *Get h* operation will only continue when $h :: s \rightarrow k$ is given a variable of type *s*.

Proposition 4.8 - Semantic Domain of State

The semantic domain for *State* is a function of type

$$s \rightarrow (a, s)$$

The first *s* is the old state & the second is the new state.

This is a carrier for stateful computations.

Definition 4.14 - evalState

```

evalState :: Free (State s) a → (s → (a, s))
evalState = eval alg gen
  where
    gen :: a → (s → (a, s))
    gen x = λs → (x, s)
    alg :: State s (s → (a, s)) → (s → (a, s))
    alg (Put s' k) = λs → k s'
    alg (Get k)    = λs → k s s

```


Here *s'* is the state we want to go to, since *Put* is our only change to change state we ignore *s* in favour of *s'*.


In *Get's* $k s s$ the first *s* generates the program & the second is passed onto future programs.


4.6 Diagram of Operations

Definition 4.15 - Convention for Diagrams

Here are the common conventions for syntax tree diagrams

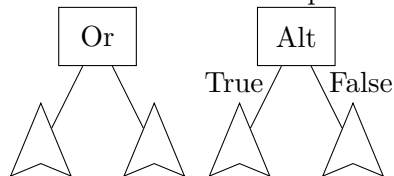
-  Operations denoted by boxes

-  Variables denoted by triangles

-  Arbitrary syntax tree by broken triangle

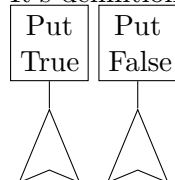
Proposition 4.9 - Or & Alt Tree Diagrams

The non-determinism operations *Or* and *Alt* can be represented by


Proposition 4.10 - Put Diagrams

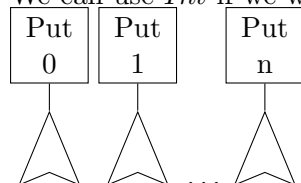
The state operation *Put* can be defined with different data types.

It's definition as *Put Bool* as



Using *Bool* only allows us to define two states.

We can use *Int* if we want more

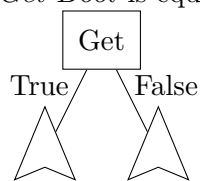


N.B. - These syntax trees corresponds to values of type *Free (State s) a*.

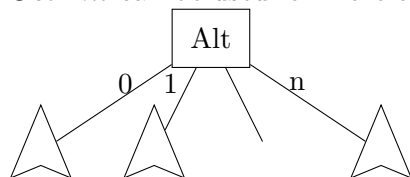
Proposition 4.11 - Get Diagrams

The *Get* operation is a generalisation of the *Alt* operation.

Get Bool is equivalent to *Alt*



Get Int can be used for more options.



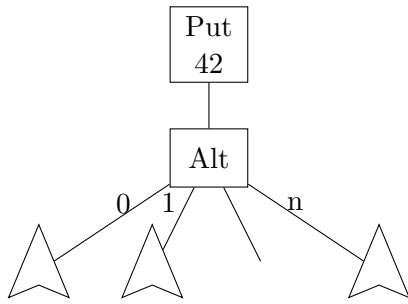
N.B. - These syntax trees corresponds to values of type *Free (State s) a*.

Remark 4.9 - Alt Bool & Get Bool

Alt Bool and *Get Bool* are not syntactially similar.

They do share the same structure.

We can compose the trees together as



N.B. - This syntax tree corresponds to values of type $Free (State\ s)\ a$.

Remark 4.10 - *Motivation for smart constructors for $Free (State\ s)\ a$*

The above syntax trees all correspond to the values of type $Free (State\ s)\ a$.

They can be cumbersome to work with as they must be wrapped in a series of *Ops*

$$Op(Put\ False\ (Op\ (Get\ (\lambda s \rightarrow \dots))))$$

We can avoid this by introducing smart constructors for *Put* and *Get*.

Definition 4.16 - *Put Smart Constructor*

Here is a definition for a smart constructor of *Put*.

```
put :: stFree (State s) ()
put s = Op (Put s (Var ()))
```

Definition 4.17 - *Get Smart Constructor*

Here is a definition for a smart constructor for *Get*

```
get :: Free (State s) s
get = Op (Get (\s \Var s))
```

Proposition 4.12 - *Using Smart Constructors*

With these smart constructor definitions we can use substitution

```
put False >> get
```

The above expression is equivalent to that in **Remark 4.10**

5 Semantics of Programming Languages

Remark 5.1 - *Motivation*

We want a clean mathematical definition of program semantics that does not rely on informal intuitions or obscure translations.

This is needed to allow transparent formal reasoning about programming languages in order to facilitate

- Program Verification;
- Compiler Construction & Optimisation;
- Language Prototyping *etc.*

Proposition 5.1 - *Types of Semantics*

There are three categories for semantics, each of which has its own sub-categories

1. Operational

- (a) Structural $\langle S, \sigma \rangle \rightarrow \langle S', \sigma' \rangle$;

(b) Natural $\langle S, \sigma \rangle \rightarrow \sigma'$.

2. Denotational

(a) Direct $\mathcal{S}[[S]]\sigma = \sigma'$;

(b) Continuation $\mathcal{S}[[S]]c\sigma = \sigma'$.

3. Axiomatic

(a) Partial $\{P\}SQ$;

(b) Total $\{P\}S\{\Downarrow Q\}$.

Remark 5.2 - Use of Semantics

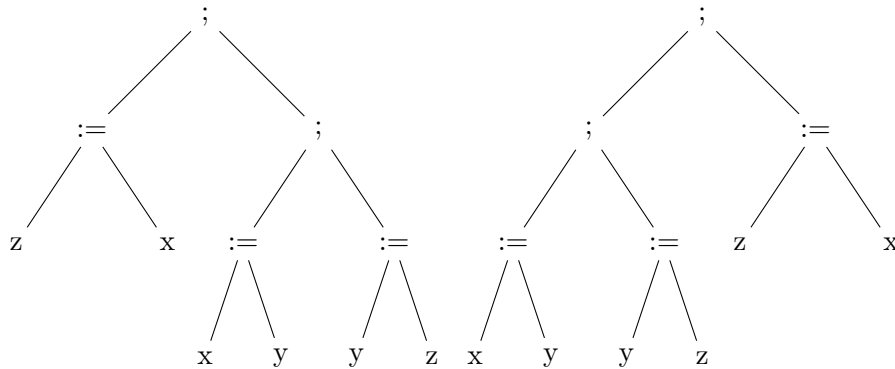
Semantics are used to assign meaning to abstract syntax trees.

Using abstract syntax trees resolves any ambiguity that may be present in the abstract syntax of a language.

Proposition 5.2 - Unambiguity of Abstract Syntax Tree

Consider the Expressions $z := x; (x := y; y := z)$ and $(z := x; x := y); y := z$.

It is not necessarily obvious that these expression are equivalent, but by considering the abstract syntax trees of each it becomes obvious.



Definition 5.1 - Semantic Functions

Semantic Functions are functions we define before looking at the semantics of program statements.

These capture the meaning of simple expressions.

Semantic Functions should be compositional in the sense that they define the meaning of an expression in terms of the meaning of sub-expressions.

Proposition 5.3 - Denoting Semantic Functions

Semantic Functions are denoted by curly letters with square brackets.

Let X be a syntactic category & Y be a semantic class, then a *Semantic Function* can be denoted as

$$\mathcal{F}[[.]] : X \rightarrow Y$$

Anything inside the brackets is syntax and everything outside is semantics.

Definition 5.2 - Compositional Definition

A *Compositional Definition* is a definition of a semantics in terms of a function.

Compositional Definitions have a *semantic clause* for each of the basis elements of the syntactic category and for each of the methods for constructing composite elements.

The *semantic clauses* for composite elements are defined in terms of the semantics of the immediate constituents of the elements.

Example 5.1 - Compositional Definition, Binary Numbers

The follow is a definition of a semantic function which return the decimal value of a binary number.

Syntax	n	$:=$	$0 1 n\ 0 n\ 1$
Semantics	\mathcal{N}	$:$	$\text{Num} \rightarrow \mathbb{Z}$
	$\mathcal{N}[[0]]$	$=$	0
	$\mathcal{N}[[1]]$	$=$	1
	$\mathcal{N}[[n\ 0]]$	$=$	$2^* \mathcal{N}[[n]]$
	$\mathcal{N}[[n\ 1]]$	$=$	$1 + 2^* \mathcal{N}[[n]]$

$$\begin{aligned}
 \text{Example } \mathcal{N}[[101]] &= 1 + 2^* \mathcal{N}[[10]] \\
 &= 1 + 2^* (2^* \mathcal{N}[[1]]) \\
 &= 1 + 2^* (2^* 1) \\
 &= 5
 \end{aligned}$$

Definition 5.3 - Program State

A *Program State* is a function from variables to integers.

We use subscript notation to denote a state which maps particular variables x_k to particular integers i_k st

$$s_{x_1=i_1, \dots, x_m=i_m} = \{(x_1, i_1), \dots, (x_m, i_m)\}$$

Program States are used for more complex semantic functions of the form

$$\mathcal{F}[[.]] : X \rightarrow (\text{State} \rightarrow Y)$$

Example 5.2 - Arithmetic Expressions & Program State

The following is an example of how program state affects the value of an arithmetic expressions

Syntax	a	$:=$	$n x a_1+a_2 a_1*a_2 a_1-a_2$
Semantics	\mathcal{A}	$:$	$\text{Aexp} \rightarrow (\text{State} \rightarrow \mathbb{Z})$
	$\mathcal{A}[[n]]s$	$=$	$\mathcal{N}[[n]]$
	$\mathcal{A}[[x]]s$	$=$	sx
	$\mathcal{A}[[a_1 + a_2]]s$	$=$	$\mathcal{A}[[a_1]]s + \mathcal{A}[[a_2]]s$
	$\mathcal{A}[[a_1 * a_2]]s$	$=$	$\mathcal{A}[[a_1]]s * \mathcal{A}[[a_2]]s$
	$\mathcal{A}[[a_1 - a_2]]s$	$=$	$\mathcal{A}[[a_1]]s - \mathcal{A}[[a_2]]s$

$$\begin{aligned}
 \text{Example } \mathcal{A}[[x + 1]]_{s_{x=3}} &= \mathcal{A}[[x]]_{s_{x=3}} + \mathcal{A}[[1]]_{s_{x=3}} \\
 &= s_{x=3}x + \mathcal{N}[[1]] \\
 &= 3 + 1 \\
 &= 4
 \end{aligned}$$

Definition 5.4 - Equivalence

Two arithmetic expressions a & b are equivalent iff they have the same semantics, $\mathcal{A}[[a]] = \mathcal{A}[[b]]$. This means that $\forall s \mathcal{A}[[a]]s = \mathcal{A}[[b]]s$.

Definition 5.5 - Total Function

A function which has a mapping for every element of the input is called a *Total Function*.

Formal

A function $f : X \rightarrow Y$ is total if $\forall x \in X \exists y \in Y \text{ st } f(x) = y$.

N.B. A *Total Function* is denoted with a straight arrow, $f : X \rightarrow Y$.

Definition 5.6 - Partial Function

A function which has a mapping for some elements of the input is called a *Partial Function*.

Formal

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

N.B. A *Partial Function* is denoted with a hooked arrow, $f : X \hookrightarrow Y$.

Remark 5.3 - Graph & Maplet Notation

A function can be defined by its *graph* as

$$f = \{(x_1, y_1), \dots, (x_n, y_n)\}$$

which is often written using *maplet* notation

$$f = \{(x_1 \mapsto y_1), \dots, (x_n \mapsto y_n)\}$$

Definition 5.7 - Structural Induction

The process for *Structural Induction* is defined by

- i) Prove that the property holds for all the *basis elements* of the syntactic category.
- ii) Prove that the property holds for all *composite elements* of the syntactic category: Assume that the property holds for all the immediate constituents of the element and prove that it also holds for the element itself.

Example 5.3 - Structural Induction, \mathcal{N}

Consider the function \mathcal{N} defined in **Example 5.1**.

Here we shall do a *Structural Induction* to prove it is a total function.

Base Cases

- If $n = 0$ then $\exists! a \in \mathbb{Z}$ st $\mathcal{N}[[n]] = a$ as the first rule of \mathcal{N} applies and so $a = 0$.
- If $n = 1$ then $\exists! a \in \mathbb{Z}$ st $\mathcal{N}[[n]] = a$ as the second rule of \mathcal{N} applies and so $a = 1$.

Composite Cases

- If $n = n'0$ then $\exists! b \in \mathbb{Z}$ st $\mathcal{N}[[n']] = b$ by the inductive hypothesis.
 $\implies \exists! a \in \mathbb{Z}$ st $\mathcal{N}[[n]] = a$ as the third rule of \mathcal{N} applies and so $a = 2 \times b$ using the fact that integer multiplication is a total function.
- If $n = n'1$ then $\exists! b \in \mathbb{Z}$ st $\mathcal{N}[[n']] = b$ by the inductive hypothesis.
 $\implies \exists! a \in \mathbb{Z}$ st $\mathcal{N}[[n]] = a$ as the fourth rule of \mathcal{N} applies and so $a = 2 \times b + 1$ using the fact that integer multiplication and addition are total functions.

5.1 Properties of the Semantics

Definition 5.8 - Free Variables

Free Variables are the set of variables that occur in an arithmetic expression.

Here is a semantic definition FV which return the *Free Variables* of the arithmetic expression a

Semantics	$FV(n)$	$= \emptyset$
	$FV(x)$	$= \{x\}$
	$FV(a_1 + a_2)$	$= FV(a_1) \cup FV(a_2)$
	$FV(a_1 \times a_2)$	$= FV(a_1) \cup FV(a_2)$
	$FV(a_1 - a_2)$	$= FV(a_1) \cup FV(a_2)$
Example	$FV(x + y \times x)$	$= FV(x) \cup FV(y \times x)$
		$= \{x\} \cup FV(y) \cup FV(x)$
		$= \{x\} \cup \{y\} \cup \{x\}$
		$= \{x, y\}$

Theorem 5.1 - Equivalent States & Arithmetic Statements

Let s, s' be two states satisfying that $s \models x = s' \models x \forall x \in FV(a)$. Then

$$\mathcal{A}[[a]]s = \mathcal{A}[[a]]s'$$

Definition 5.9 - Substitutions

Substitution is replacing a variable in an arithmetic sequence with another arithmetic expression.

Here is a semantic definition of substitution

$$\begin{aligned}
 \text{Semantics} \quad n[y \mapsto a_0] &= n \\
 x[y \mapsto a_0] &= \begin{cases} a_0 & \text{if } x = y \\ x & \text{if } x \neq y \end{cases} \\
 (a_1 + a_2)[y \mapsto a_0] &= (a_1[y \mapsto a_0]) + (a_2[y \mapsto a_0]) \\
 (a_1 \times a_2)[y \mapsto a_0] &= (a_1[y \mapsto a_0]) \times (a_2[y \mapsto a_0]) \\
 (a_1 - a_2)[y \mapsto a_0] &= (a_1[y \mapsto a_0]) - (a_2[y \mapsto a_0]) \\
 \text{Example} \quad (x + 1)[x \mapsto 3] &= (x[x \mapsto 3]) + (1[x \mapsto 3]) \\
 &= 3 + 1 \\
 &= 4
 \end{aligned}$$

Definition 5.10 - Substitution of States

There is a separate definition for substitution with states

$$(s[y \mapsto v])x = \begin{cases} v & \text{if } x = y \\ s \ x & \text{if } x \neq y \end{cases}$$

N.B. This just changes the substitutions the state makes & not the expression it is applied to.

6 Operational Semantics

6.1 Natural Operational Semantics

Definition 6.1 - Operational Semantics

In an *operational semantics* we are concerned with *how* to execute programs and not merely what the results of the execution are.

Moreover, we are interested in how the states are modified during the execution of the statement.

Proposition 6.1 - Approaches to Operational Semantics

There are two different approaches to *Operational Semantics*

- i) *Structural Operation Semantics* - Its purpose is to describe how the *individual steps* of the computations take place.
- ii) *Natural Semantics* - Its purpose is to describe how the *overall results* of executions are obtained.

Definition 6.2 - Transition System

A *Transition System* is a specification of the meaning of statements.

Transition Systems have two of configurations

- i) $\langle S, \sigma \rangle$, representing that the statement S is to be executed from the state σ ; and
- ii) σ , representing a terminal state.

Example 6.1 - Transition Systems

Here are some definitions for semantics

$$\begin{array}{lcl}
[ass_{ns}] & \overline{\langle x := a, \sigma \rangle \rightarrow \sigma[x \mapsto \mathcal{A}[[a]]\sigma]} & \\
[skip_{ns}] & \overline{\langle skip, \sigma \rangle \rightarrow \sigma} & \\
[comp_{ns}] & \frac{\overline{\langle S_1, \sigma \rangle \rightarrow \sigma'} \quad \overline{\langle S_2, \sigma' \rangle \rightarrow \sigma''}}{\overline{\langle S_1; S_2, \sigma \rangle \rightarrow \sigma''}} & \\
[if_{ns}^{tt}] & \frac{\overline{\langle S_1, \sigma \rangle \rightarrow \sigma'}}{\overline{\langle \text{if } b \text{ then } S_1 \text{ else } S_2, \sigma \rangle \rightarrow \sigma'}} & \text{if } \mathcal{B}[[b]]\sigma = tt \\
[if_{ns}^{ff}] & \frac{\overline{\langle S_2, \sigma \rangle \rightarrow \sigma'}}{\overline{\langle \text{if } b \text{ then } S_1 \text{ else } S_2, \sigma \rangle \rightarrow \sigma'}} & \text{if } \mathcal{B}[[b]]\sigma = ff \\
[while_{ns}^{tt}] & \frac{\overline{\langle S, \sigma \rangle \rightarrow \sigma'} \quad \overline{\langle \text{while } b \text{ do } S, \sigma' \rangle \rightarrow \sigma''}}{\overline{\langle \text{while } b \text{ do } S, \sigma \rangle \rightarrow \sigma''}} & \text{if } \mathcal{B}[[b]]\sigma = tt \\
[while_{ns}^{ff}] & \overline{\langle \text{while } b \text{ do } S, \sigma \rangle \rightarrow \sigma} & \text{if } \mathcal{B}[[b]]\sigma = ff
\end{array}$$

Example 6.2 - Transition Systems, Binary Digits

Here we give a definition for \mathcal{N} in terms of transition systems

$$\frac{\langle n, \sigma \rangle \rightarrow i, \langle b, \sigma \rangle \rightarrow j}{\langle n \ b, \sigma \rangle \rightarrow 2i + j}$$

Definition 6.3 - Semantic Equivalence

Two statements S_1 & S_2 are *semantically equivalent* (under the natural operational semantics) if

$$\langle S_1, \sigma \rangle \rightarrow \sigma' \Leftrightarrow \langle S_2, \sigma \rangle \rightarrow \sigma' \quad \forall \sigma, \sigma' \in \text{State}$$

Definition 6.4 - Deterministic Semantics

A natural semantics is *deterministic* if

$$\langle S, \sigma \rangle \rightarrow \sigma' \ \& \ \langle S, \sigma \rangle \rightarrow \sigma'' \implies \sigma' = \sigma'' \quad \forall \sigma, \sigma', \sigma'' \in \text{State}, S \in \text{Stm}$$

Proposition 6.2 - Induction on the Shape of Derivation Trees

Since *derivation trees* are compositionally defined, we can perform inductive reasoning over them.

This can be used to prove that semantic properties hold even when the semantics is not strictly compositional.

- i) Prove that the property holds for all the simple derivation trees by showing that it holds for the *axioms* of the transitional system.
- ii) Prove that the property holds for all composite derivation trees:
for each *rule* assume that the property holds for its premises and prove that it also holds for the conclusion of the rule provided that the conditions of the rule are satisfied.

N.B. Here the *axioms* are Assignment, Skip & While False.

The *rules* are Composition, If True, If False, While True.

Proof 6.1 - Deterministic Semantics

Show $\langle S, \sigma \rangle \rightarrow \sigma' \ \& \ \langle S, \sigma \rangle \rightarrow \sigma'' \implies \sigma' = \sigma'' \quad \forall \sigma, \sigma', \sigma'' \in \text{State}, S \in \text{Stm}$.

Now let $\sigma, \sigma', \sigma''$ be arbitrary states.

Show $\langle S, \sigma \rangle \rightarrow \sigma' \ \& \ \langle S, \sigma \rangle \rightarrow \sigma'' \implies \sigma' = \sigma'' \quad \forall S \in \text{Stm}$.

Show $\langle S, \sigma \rangle \rightarrow \sigma' \implies (\langle S, \sigma \rangle \rightarrow \sigma'' \implies \sigma' = \sigma'') \forall S \in \text{Stm}$.

Now let $\text{Stm}(\sigma, \sigma')$ be the set of statements such that $\langle S, \sigma \rangle \rightarrow \sigma'$.

Show $\langle S, \sigma \rangle \rightarrow \sigma'' \implies \sigma' = \sigma'' \forall S \in \text{Stm}(\sigma, \sigma')$.

Now, let $\text{Tree}(\sigma, \sigma')$ be the set of trees with root $\langle S, \sigma \rangle \rightarrow \sigma'$ for a $S \in \text{Stm}$.

Show $\langle S, \sigma \rangle \rightarrow \sigma'' \implies \sigma' = \sigma'' \forall T_S \in \text{Tree}(\sigma, \sigma')$.

By induction on the shape of the derivation tree T_S where for each axiom we must show $\langle S, \sigma \rangle \rightarrow \sigma'' \implies \sigma' = \sigma''$.

Assignment T_S must be $\overline{\langle x := a, \sigma \rangle \rightarrow \sigma[x \mapsto \mathcal{A}[[a]]\sigma]}$ so $S \equiv x := a$ and $\sigma' \equiv \sigma[x \mapsto \mathcal{A}[[a]]\sigma]$.
Assuming $\langle S, \sigma \rangle \rightarrow \sigma''$ only one rule, assignment, could have derived this.
But this means $\sigma'' \equiv \sigma[x \mapsto \mathcal{A}[[a]]\sigma]$ and so $\sigma' = \sigma''$.

Skip T_S must be $\overline{\langle \text{skip}, \sigma \rangle \rightarrow \sigma}$ so $S \equiv \text{skip}$ and $\sigma' \equiv \sigma$.
Assuming $\langle S, \sigma \rangle \rightarrow \sigma''$ only one rule, skip, could have derived this.
But this means $\sigma'' \equiv \sigma$ thus $\sigma' = \sigma''$.

While False T_S must be $\overline{\langle \text{While } b \text{ Do } S, \sigma \rangle \rightarrow \sigma}$ and $\mathcal{B}[[b]]\sigma = ff$ so $S \equiv \text{While } b \text{ Do } S$ and $\sigma' \equiv \sigma$.
Assuming $\langle S, \sigma \rangle \rightarrow \sigma''$ only one rule, while false, could have derived this.
But this means $\sigma'' \equiv \sigma$ thus $\sigma' = \sigma''$.

By Induction on the shape of the derivation tree T_S where for each rule we must show $\langle S, \sigma \rangle \rightarrow \sigma'' \implies \sigma' = \sigma''$.

Composition T_S must be $\overline{\overline{\langle S_1, \sigma \rangle \rightarrow \sigma'} \overline{\langle S_2, \sigma' \rangle \rightarrow \sigma''}} \overline{\langle S_1; S_2, \sigma \rangle \rightarrow \sigma''}$ so $S \equiv S_1; S_2$.
Assuming $\langle S, \sigma \rangle \rightarrow \sigma''$ only one rule, composition, could have derived this.
So, by the inductive hypothesis $\sigma_1 = \sigma_2$ thus $\sigma' = \sigma''$.

Analogous arguments apply.

6.2 Structural Operational Semantics

Remark 6.1 - Order of Argument Evaluation

The order in which arguments evaluated in operations depends on the how the semantics of the operations are defined.

The most common are *left-first*, *right-first* & *parallel*.

N.B. Consider how *if* statements evaluate their conditions.

Remark 6.2 - Alternative Name

Structural Operation Semantics are occasionally referred to as *Small Step Semantics*.

Proposition 6.3 - Overview

Structural Operation Semantics is concerned with *how* a program performs a computation, rather than *which* computation it ultimately performs.

Structural Operation Semantics gives a finer level of control over the order of argument evaluation.

Proposition 6.4 - Structural Approach

The *Structural Approach* is to progressively reduce intermediate configurations of $\langle E, \sigma \rangle$ step-by-step until a semantic value v is finally reached.

6.2.1 Configurations

Definition 6.5 - Configuration Forms

A configuration γ can be one of two forms

- i) *Incomplete* when $\gamma = \langle S, \sigma \rangle$; (aka *intermediate*, or

ii) *Terminal* when $\gamma = \sigma$. (aka *intermediate*)

Definition 6.6 - Configuration Properties

A configuration γ can have one of two properties

i) *Stuck* if $\nexists \gamma'$ st $\gamma \Rightarrow \gamma'$; or,

ii) *Unstuck* if $\exists \gamma'$ st $\gamma \Rightarrow \gamma'$.

Definition 6.7 - Derivation Sequence

A *Derivation Sequence* from $\gamma_0 = \langle S, \sigma \rangle$ is defined as either

i) A *Finite* sequence $\gamma_0, \dots, \gamma_n$ st $\gamma_i \Rightarrow \gamma_{i+1} \forall 0 \leq i \leq n-1$ and γ_n is a terminal or stuck configuration; Or,

ii) A *Infinite* sequence $\gamma_0, \gamma_1, \dots$ st $\gamma_i \Rightarrow \gamma_{i+1} \forall 0 \leq i$.

Notation 6.1 - Steps in Sequence

$\gamma \Rightarrow^k \gamma'$ denotes that γ' can be obtained from γ in *exactly* k steps using the \Rightarrow transition relation.

$\gamma \Rightarrow^* \gamma'$ denotes that γ' can be obtained from γ in *some* finite number of steps, using the \Rightarrow transition relation.

Definition 6.8 - Terminating Execution

The execution of a statement S in state σ *terminates* iff \exists a *Finite Derivation Sequence* from $\langle S, \sigma \rangle$.

S *always terminates* iff its execution terminates in all states σ .

Definition 6.9 - Looping Execution

The execution of a statement S in state σ *loops* iff \exists an *Infinite Derivation Sequence* from $\langle S, \sigma \rangle$.

S *always loops* iff its execution loops in all states σ .

Definition 6.10 - Successful Execution

The execution of statement S in state σ *terminates successfully* iff it ends with a terminal configuration.

6.2.2 Semantic Definitions

Definition 6.11 - Deterministic

A structural operation semantics is *Deterministic* iff

$$\langle S, \sigma \rangle \Rightarrow \gamma \ \& \ \langle S, \sigma \rangle \Rightarrow \gamma' \implies \gamma = \gamma' \ \forall \ S, \sigma, \gamma, \gamma'$$

Definition 6.12 - Weakly Deterministic

A structural operation semantics is *Deterministic* iff

$$\langle S, \sigma \rangle \Rightarrow^* \sigma' \ \& \ \langle S, \sigma \rangle \Rightarrow^* \sigma'' \implies \sigma' = \sigma'' \ \forall \ S, \sigma, \sigma', \sigma''$$

Definition 6.13 - Semantic Equivalence

Two statements S_1 & S_2 are *Semantically Equivalent* under structural semantics whenever it holds that $\forall \sigma \in State$

- $\langle S_1, \sigma \rangle \Rightarrow^* \gamma$ iff $\langle S_2, \sigma \rangle \Rightarrow^* \gamma$ whenever γ is terminal, or stuck; And,
- There is an infinite derivation sequence from $\langle S_1, \sigma \rangle$ to γ iff there is an infinite derivation sequence from $\langle S_2, \sigma \rangle$ to γ .

Definition 6.14 - Transitions

Transitions between configurations are defined by axioms and rule schema.

Rule Instances are obtained by replacing meta-variables by types which satisfy the side conditions.

Transitions are denoted by \Rightarrow .

Proposition 6.5 - Semantics of Skip & Assignment

The semantics of *Skip* & *Assignment* are unchanged from the natural semantics as they are reduced to their final states in just one step.

The only change is in the notation, to include \Rightarrow .

$$\begin{array}{c} [skip_{SOS}] \quad \overline{\langle skip, \sigma \rangle \Rightarrow \sigma} \\ [ass_{SOS}] \quad \overline{\langle x := a, \sigma \rangle \Rightarrow \sigma[x \mapsto \mathcal{A}[[a]]\sigma]} \end{array}$$

Proposition 6.6 - Semantics of Composition

The semantics of *Composition* is different from the natural semantics since it depends on whether the first operation reduces to a complete configuration (skip or assignment), or not.

$$\begin{array}{c} [comps_{SOS}] \quad \frac{\overline{\langle S_1, \sigma \rangle \Rightarrow \sigma'}}{\langle S_1; S_2, \sigma \rangle \Rightarrow \langle S_2, \sigma' \rangle} \quad \text{if } \langle S_1, \sigma \rangle \text{ reduces to a complete state } \sigma' \\ \frac{\overline{\langle S_1, \sigma \rangle \Rightarrow \langle S'_1, \sigma' \rangle}}{\langle S_1; S_2, \sigma \rangle \Rightarrow \langle S'_1; S_2, \sigma' \rangle} \quad \text{otherwise} \end{array}$$

Proposition 6.7 - Semantics of If

For *if* we have two rules, depending upon whether the condition is true or false.

$$\begin{array}{c} [if_{SOS}^{tt}] \quad \overline{\langle \text{if } b \text{ then } S_1 \text{ else } S_2, \sigma \rangle \Rightarrow \langle S_1, \sigma \rangle} \quad \text{if } \mathcal{B}[[b]]\sigma = tt \\ [if_{SOS}^{ff}] \quad \overline{\langle \text{if } b \text{ then } S_1 \text{ else } S_2, \sigma \rangle \Rightarrow \langle S_2, \sigma \rangle} \quad \text{if } \mathcal{B}[[b]]\sigma = ff \end{array}$$

Proposition 6.8 - Semantics of While Loop

For *while* we have two rules, depending upon whether the conditional is true or false

$$\begin{array}{c} [while_{SOS}^{tt}] \quad \overline{\langle \text{while } b \text{ do } S, \sigma \rangle \Rightarrow \langle S; \text{while } b \text{ do } S, \sigma \rangle} \quad \text{if } \mathcal{B}[[b]]\sigma = tt \\ [while_{SOS}^{ff}] \quad \overline{\langle \text{while } b \text{ do } S, \sigma \rangle \Rightarrow \sigma} \quad \text{if } \mathcal{B}[[b]]\sigma = ff \end{array}$$

Example 6.3 - Variable Swap

We can prove the following is a valid *Derivation Sequence*

$$\langle (z := x, x := y); y := z, \sigma_{570} \rangle \Rightarrow \langle x := y; y := z, \sigma_{575} \rangle \Rightarrow \langle y := \sigma_{775} \rangle \Rightarrow \sigma_{755}$$

using the following derivation trees

$$\begin{array}{c} \overline{\langle z := x, \sigma_{570} \rangle \Rightarrow \sigma_{575}} \\ \overline{\langle z := x; x := y, \sigma_{570} \rangle \Rightarrow \langle x := y, \sigma_{575} \rangle} \\ \overline{\langle (z := x; x := y); y := z, \sigma_{570} \rangle \Rightarrow \langle x := y; y := z, \sigma_{575} \rangle} \\ \overline{\langle x := y, \sigma_{575} \rangle \Rightarrow \sigma_{775}} \\ \overline{\langle x := y; y := z, \sigma_{575} \rangle \Rightarrow \langle y := z, \sigma_{775} \rangle} \\ \overline{\langle y := z, \sigma_{775} \rangle \Rightarrow \sigma_{755}} \end{array}$$

Definition 6.15 - Semantic Function

For a deterministic structural operational semantics, we can define a *Semantic Function* by

- $S_{SOS}[[.]] : Stm \rightarrow (State \hookrightarrow State)$.
- $S_{SOS}[[S]]\sigma = \begin{cases} \sigma' & \text{if } \langle S, \sigma \rangle \Rightarrow^* \sigma' \\ \text{undefined} & \text{otherwise} \end{cases}$.

Proposition 6.9 - *Semantic Function, Non-Deterministic*

For a non-deterministic semantics we can use the following definition

- $S'_{SOS}[[.]] : Stm \hookrightarrow (State \hookrightarrow State)$ to ignore ambiguous cases; or,
- $S''_{SOS}[[.]] : Stm \rightarrow (State \rightarrow 2^{State})$ to allow a set of final states; or,
- $S'''_{SOS}[[.]] : Stm \rightarrow (2^{State} \rightarrow 2^{State})$ to facilitate function composition.

Theorem 6.1 - *Natural v Operation Semantic Results*

$$\forall S \in Stm \ S_{ns}[[S]] = S_{SOS}[[S]]$$

Theorem 6.2 - *Induction on the Length of Derivation Sequences*

The process for *Induction on the Length of Derivation Sequences*

- i) Prove that the property holds for all derivation sequences of length 0.
- ii) Prove that the property holds for all other derivation sequences:
 - (a) Assume that the property holds for all derivation sequence of length at most k ;
 - (b) Show that it holds for all derivation sequences of length $k + 1$.

N.B. This is just a weak induction.

6.3 Provably Correct Implementation

Remark 6.3 - *Motivation*

When given a formal specification for the semantics of a programming language it becomes possible to argue about the correctness of its implementation.

Here the *While* language is translated into a structured form of assembler code for an abstract machine.

6.3.1 The Abstract Machine

Definition 6.16 - *Configurations*

The *Abstract Machine* has *Configurations* of the form $\langle c, e, s \rangle$ where

- c is the code to be executed; ($c \in Code = Inst$).
- e is the evaluation stack of expressions; and, ($e \in (\mathbb{Z} \cup T)$).
- s is the storage for variables. ($s \in State = Var \rightarrow \mathbb{Z}$).

Definition 6.17 - *Instructions*

The *Instructions* for the *Abstract Machine* are given by the abstract syntax

```

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

```

Notation 6.2 - Transition \triangleright

The transition relation \triangleright specifies how to execute the instructions.

$\gamma \triangleright^k \gamma'$ means γ' can be obtained from γ in exactly k steps.

$\gamma \triangleright^* \gamma'$ means γ' can be obtained from γ in some finite number of steps.

Definition 6.18 - 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	$\begin{cases} \langle c_1 : c, e, s \rangle & \text{if } t = tt \\ \langle c_2 : c, e, s \rangle & \text{if } t = ff \end{cases}$	
$\langle \text{LOOP}(c_1, c_2) : c, e, s \rangle$	\triangleright	$\langle c_1 : \text{BRANCH}(c_2 : \text{LOOP}(c_1, c_2), \text{NOOP}) : c, e, s \rangle$	

Definition 6.19 - Configuration Form

A configuration γ can have one of two forms

- i) *Incomplete* when $\gamma = \langle c : cs, e, \sigma \rangle$; or
- ii) *Terminal* when $\gamma = \langle \varepsilon, e, \sigma \rangle$.

Definition 6.20 - Configuration Properties

An incomplete configuration γ can have two properties

- i) *Stuck* if $\nexists \gamma'$ st $\gamma \triangleright \gamma'$; or,
- ii) *Unstuck* if $\exists \gamma'$ st $\gamma \triangleright \gamma'$

Definition 6.21 - Computation Sequence

A *Computation Sequence* can be either

- i) *Finite* sequence $\gamma_0, \dots, \gamma_n$ st $\gamma_0 = \langle c, \varepsilon, \sigma \rangle$ and $\gamma_i \triangleright \gamma_{i+1} \ \forall i \in [0, n-1]$ & γ_n is a terminal configuration; or,
- ii) *Infinite* sequence $\gamma_0, \gamma_1, \dots$ st $\gamma_0 = \langle c, \varepsilon, \sigma \rangle$ and $\gamma_i \triangleright \gamma_{i+1} \ \forall i \in \mathbb{N}$.

Proposition 6.10 - Induction on the Length of Computation Sequence

Here is a technique to prove a property holds, regardless of the length of a computation sequence

- i) Prove that the property holds for all computation sequences of length 0.
- ii) Prove that the property holds for all other computation sequence:
 - (a) Assume that the property holds for all computation sequences of length at most k .
 - (b) Show that it holds for computation sequences of length $k+1$.

Example 6.4 - Terminating

- $\langle \text{PUSH-1 : FETCH-}x : \text{ADD : STORE-}x, \varepsilon, s_{x=3} \rangle$
- $\triangleright \langle \text{FETCH-}x : \text{ADD : STORE-}x, \varepsilon, s_{x=3} \rangle$
- $\triangleright \langle \text{ADD : STORE-}x, 3 : 1, s_{x=3} \rangle$
- $\triangleright \langle \text{STORE-}x, 4, s_{x=3} \rangle$
- $\triangleright \langle \varepsilon, \varepsilon, s_{x=4} \rangle$

Example 6.5 - Looping

- $\langle \text{LOOP (TRUE , NOOP)}, \varepsilon, s \rangle$
- $\triangleright \langle \text{TRUE : BRANCH (NOOP : LOOP (TRUE , NOOP), NOOP)}, \varepsilon, s \rangle$
- $\triangleright \langle \text{BRANCH (NOOP : LOOP (TRUE , NOOP), NOOP)}, \varepsilon, s \rangle$
- $\triangleright \langle \text{NOOP : LOOP (TRUE , NOOP)}, \varepsilon, s \rangle$
- $\triangleright \langle \text{LOOP (TRUE , NOOP)}, \varepsilon, s \rangle$
- $\triangleright \dots$

Definition 6.22 - Deterministic Semantics

The *Semantics* of an *Abstract Machine* is *Deterministic* iff $\forall \gamma, \gamma', \gamma''$ we have that

$$\gamma \triangleright \gamma' \ \& \ \gamma \triangleright \gamma'' \implies \gamma' = \gamma''$$

Definition 6.23 - Execution Function

We define the *Execution Function* for *Abstract Machines* as

$$\mathcal{M}[[\cdot]] : \text{Code} \rightarrow (\text{State} \rightarrow \text{State})$$

$$\mathcal{M}[[c]]\sigma = \begin{cases} \sigma' & \text{if } \langle c, \varepsilon, \sigma \rangle \triangleright^* \langle \varepsilon, e, \sigma' \rangle \\ \text{Undefined} & \text{Otherwise} \end{cases}$$

Proposition 6.11 - Code Translation of Arithmetic Expressions

We define $\mathcal{CA} : Aexp \rightarrow \text{Code}$ to be a total function that translates arithmetic expressions from the *While* language to the *Code* of *Abstract Machines* language.

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

Proposition 6.12 - Code Translation of Boolean Expressions

We define $\mathcal{CB} : Bexp \rightarrow \text{Code}$ to be a total function that translates boolean expressions from the *While* language to the *Code* of *Abstract Machines* language.

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

Proposition 6.13 - Code Translation of Boolean Expressions

We define $\mathcal{CS} : Stm \rightarrow \text{Code}$ to be a total function that translates statements from the *While* language to the *Code* of *Abstract Machines* language.

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

Definition 6.24 - The Semantic Function, \mathcal{S}_{am}

The meaning of a statement S can be obtained by translating it into code for *Abstract Machine*

and then executing the code on the abstract machine.

The effect of this is expressed by the *Semantic Function*

$$\mathcal{S}_{am} : Stm \rightarrow (State \hookrightarrow State)$$

define by

$$\mathcal{S}_{am}[[S]] = (\mathcal{M} \cdot \mathcal{CS})[[S]]$$

6.3.2 Correctness of Implementation

Remark 6.4 - Proving Correctness of Implementation

Proving Correctness of Implementation amounts to showing that if we translate a statement into code for *Abstract Machine* & then execute that code, then we must obtain the sample result as specified by the operational semantics of *While*.

Theorem 6.3 - Correctness of Arithmetic Expressions

For all arithmetic expressions a we have

$$\langle \mathcal{CA}[[a]], \varepsilon, s \rangle \triangleright^* \langle \varepsilon, \mathcal{A}[[a]]s, s \rangle$$

N.B. All intermediate configurations of this computation sequence will have a non-empty evaluation stack.

Proof 6.2 - Correctness of Arithmetic Expressions

This is a proof by structural induction on a .

Case - n .

$$\langle \mathcal{CA}[[n]], \varepsilon, s \rangle = \langle PUSH - n, \varepsilon, s \rangle \triangleright \langle \varepsilon, \mathcal{N}[[n]], s \rangle = \langle \varepsilon, \mathcal{A}[[n]]s, s \rangle.$$

Case - x .

$$\langle \mathcal{CA}[[x]], \varepsilon, s \rangle = \langle FETCH - n, \varepsilon, s \rangle \triangleright \langle \varepsilon, (s \ x), s \rangle = \langle \varepsilon, \mathcal{A}[[x]]s, s \rangle.$$

Case - $a_1 + a_2$.

$$\langle \mathcal{CA}[[a_1 + a_2]], \varepsilon, s \rangle = \langle \mathcal{CA}[[a_2]] : \mathcal{CA}[[a_1]] : ADD, \varepsilon, s \rangle.$$

By the induction hypothesis $\langle \mathcal{CA}[[a_i]], \varepsilon, s \rangle \triangleright^* \langle \varepsilon, \mathcal{A}[[a_i]]s, s \rangle$ for $i \in \{1, 2\}$.

Thus

$$\begin{aligned} \langle \mathcal{CA}[[a_2]] : \mathcal{CA}[[a_1]] : ADD, \varepsilon, s \rangle &\triangleright^* \langle \mathcal{CA}[[a_1]] : ADD, \mathcal{A}[[a_2]]s, s \rangle \\ &\triangleright^* \langle ADD, (\mathcal{A}[[a_1]]s) : (\mathcal{A}[[a_2]]s), s \rangle \\ &\triangleright \langle \varepsilon, \mathcal{A}[[a_1]]s + \mathcal{A}[[a_2]]s, s \rangle \end{aligned}$$

Similar arguments can be made for the other compositional cases.

Thus the hypothesis holds for all cases.

Theorem 6.4 - Correctness of Boolean Expressions

For all boolean expressions b we have

$$\langle \mathcal{CB}[[b]], \varepsilon, s \rangle \triangleright^* \langle \varepsilon, \mathcal{B}[[b]]s, s \rangle$$

N.B. The proof for this is similar to that for *Correctness of Arithmetic Expressions*.

Theorem 6.5 - Correctness of Statements

For all statements S of *While* we have that

$$\mathcal{S}_{ns}[[S]] = \mathcal{S}_{am}[[S]]$$

Theorem 6.6 - Results of Termination are equals

For all statements S of *While* and states s, s' we have

$$\text{If } \langle S, s \rangle \rightarrow s' \text{ then } \langle \mathcal{CS}[[S]], \varepsilon, s \rangle \triangleright^* \langle \varepsilon, \varepsilon, s' \rangle$$

i.e. If the execution of S from s terminates in the natural semantics then the execute of code S from storage s & the results will be the same.

7 Denotational Semantics

7.1 Partial Orders & Fixpoints

Definition 7.1 - Relation

A *Relation* is any subset of a Cartesian product of two sets.

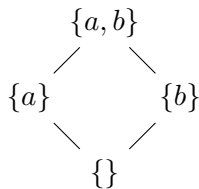
Definition 7.2 - Weak Partial Order, \leq

A *Weak Partial Order* is a relation \leq on a set S that is

- i) Reflexive, $a \leq a \forall a \in S$;
- ii) Transitive, if $a \leq b$ & $b \leq a \implies a = b$; and,
- iii) Antisymmetric, if $a \leq b$ & $b \leq c \implies a \leq c \forall a, b, c \in S$.

Example 7.1 - Weak Partial Order

Below the \subseteq relation is a *weak partial order*.



Definition 7.3 - Strong Partial Order

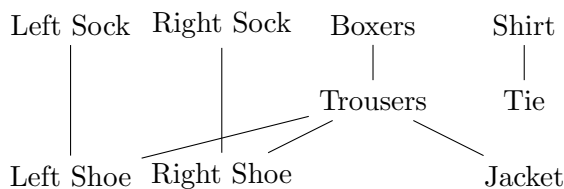
A *Strong Partial Order* is a relation \leq on a set S that is

- i) Irreflexive, $\nexists a \in S$ st $a \leq a$;
- ii) Transitive, if $a \leq b$ & $b \leq a \implies a = b$; and,
- iii) Antisymmetric, if $a \leq b$ & $b \leq c \implies a \leq c \forall a, b, c \in S$.

N.B. This is also known as a *Strict Partial Order*.

Example 7.2 - Strong Partial Order

Below the "put on before" relation is a *strong partial order*.



Definition 7.4 - Total Partial Order

A *Total Partial Order* is a relation \leq on a set S that is

- i) Connex, $a \leq b$ or $b \leq a \forall a, b \in S$;
- ii) Reflexive, $a \leq a \forall a \in S$;
- iii) Transitive, if $a \leq b$ & $b \leq c \implies a \leq c \forall a, b, c \in S$; and,
- iv) Antisymmetric, if $a \leq b$ & $b \leq a \implies a = b$.

N.B. This is also known as a *Chain*.

Example 7.3 - Total Partial Order

Below the \leq relation is a *total partial order*.

$\dots - 3 - 2 - 1 - \dots$

Definition 7.5 - Fixpoints

The *Fixpoints* of a function $f : X \rightarrow X$ are all x where $f(x) = x$.

Definition 7.6 - Least Fixpoint

The *Least Fixpoint* of a function is the lowest *Fixpoint* value.

Example 7.4 - Fixpoints & Least Fixpoints

f	$\text{fix}(f)$	$\text{lfp}(f)$
$x \mapsto x^2$	$\{0, 1\}$	0
$x \mapsto 2x$	$\{0\}$	0
$x \mapsto x + 1$	$\{\}$	undefined
$x \mapsto x$	\mathbb{R}	undefined

Proposition 7.1 - Fixpoints Haskell Implementation

To compute the least fixpoint of a function using Haskell we define the following function

```
fix f = f (fix f)
```

Example 7.5 - Fixpoints Haskell Implementation

```
fix (*2) = (*2) (fix (*2))
          = (*2) (*2) (fix (*2))
          :
          = ⊥
```

```
fix (const 2) = (const 2) (fix (const 2))
               = 2
```

Remark 7.1 - Fixpoints in Recursive Functions

Fixpoints can often be used to redefine recursive functions.

This includes factorial

```
fac = fix (\f n -> if n==0 then 1 else n * f (n-1))
```

Definition 7.7 - Trans

Trans is the set of all possible state transformers

$$\text{Trans} = (\text{State} \hookrightarrow \text{State})$$

Definition 7.8 - \sqsubseteq Relation

We define the \sqsubseteq relation over *Trans*

$$\forall g_1, g_2 \in \text{Trans}, g_1 \sqsubseteq g_2 \implies \forall s, s' \in \text{State} \text{ if } g_1 s = s' \implies g_2 s = s'$$

N.B. if $g_1 \sqsubseteq g_2$ we say “ g_1 shares its results with g_2 ”.

Example 7.6 - \sqsubseteq Relation

Consider the following definitions

$$g_1 s = s \quad \forall s \qquad g_2 s = \begin{cases} s & \text{if } s \geq 0 \\ \text{undef} & \text{otherwise} \end{cases}$$

$$g_3 s = \begin{cases} s & \text{if } s = 0 \\ \text{undef} & \text{otherwise} \end{cases} \qquad g_4 s = \begin{cases} s & \text{if } s \leq 0 \\ \text{undef} & \text{otherwise} \end{cases}$$

Then $g_3 \sqsubseteq g_2 \sqsubseteq g_1$ & $g_3 \sqsubseteq g_4 \sqsubseteq g_1$ only.

Theorem 7.1 - *Type of order, \sqsubseteq*
 \sqsubseteq is a weak partial order over *Trans*.

Proof 7.1 - *Theorem 7.1*

Reflexivity - Trivial.

Transitivity

Let g_1, g_2, g_3 be arbitrary state transformers with $g_1 \sqsubseteq g_2$ & $g_2 \sqsubseteq g_3$.

Assume $g_1 s = s'$ for arbitrary states s, s' .

$\implies g_2 s = s' \implies g_3 s = s' \implies g_1 \sqsubseteq g_3$.

Anti-Symmetry

Let g_1, g_2 be arbitrary state transformers with $g_1 \sqsubseteq g_2$ & $g_2 \sqsubseteq g_1$.

Assume $g_1 s = s' \implies g_2 s = s'$.

If $s' = \perp \implies g_1 s$ is undefined.

if $s' \neq \perp$ then $g_1 s = s' \implies g_2 s = s' \implies g_1 s = s' \implies g_1 = g_2$.

Theorem 7.2 - \sqsubseteq has a least element

\sqsubseteq has a unique least element.

This is the function that is undefined for all states.

Definition 7.9 - *Partial Order Set*

Let \sqsubseteq be a partial order on a set D .

This combination is a *Partial Order Set* (PO-Set).

It is denoted (D, \sqsubseteq) .

Definition 7.10 - *Upper Bound of PO-Set*

Let $Y \subseteq D$ be any subset.

The element $d \in D$ is called an *Upper Bound* of Y iff $t \sqsubseteq d \forall y \in Y$.

Definition 7.11 - *Least Upper Bound of Partial Order Set*

Let $Y \subseteq D$ be any subset.

The element $d \in D$ is called a least upper bound of Y iff $d \sqsubseteq d'$ for all upper bounds d' of Y .

N.B. We denote d as $\bigsqcup Y$.

Remark 7.2 - *Least Upper Bound is Union*

Generally the *Least Upper Bound* of a *Partial Order Set* is the union of all elements of the set.

Definition 7.12 - *Chain*

We say that a set $Y \subseteq D$ is a chain iff $y_1 \sqsubseteq y_2$ or $y_2 \sqsubseteq y_1 \forall y_1, y_2 \in Y$.

This means Y is a totally ordered subset of D .

Definition 7.13 - *Chain Complete Partial Orders, CCPO*

A PO-Set (D, \sqsubseteq) is called a *Chain-Complete Partially Ordered Set* if the least upper bound \bigsqcup exists for all chains $Y \subseteq D$.

Definition 7.14 - *Complete Lattice*

A PO-Set (D, \sqsubseteq) is called a *Complete Lattice* if the least upper bound \bigsqcup exists for all subsets $Y \subseteq D$.

Definition 7.15 - *Monotone Function*

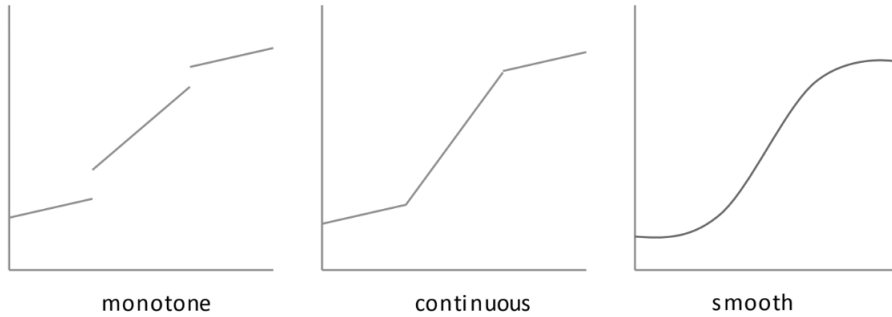
Let (D, \sqsubseteq) and (D', \sqsubseteq') be CCPOs and f be a total function with signature $f : D \rightarrow D'$.
 f is *Monotone* iff $d_1 \sqsubseteq d_2 \implies f(d_1) \sqsubseteq' f(d_2) \forall d_1, d_2 \in D$.

Definition 7.16 - Continuous Functions

Let (D, \sqsubseteq) and (D', \sqsubseteq') be CCPOs and let f be a monotone function with signature $f : D \rightarrow D'$.
 f is *Continuous* iff the following holds $\forall Y \subseteq D$ with $Y \neq \emptyset$

$$\bigsqcup \{f(d) \mid d \in Y\} = f(\bigsqcup Y)$$

Remark 7.3 - Visual Intuition for Function Analysis



Theorem 7.3 - Fixpoint Theorem

Let $f : D \rightarrow D$ be a continuous function on a *Chain Complete Partial Order* $\langle D, \sqsubseteq \rangle$ with least element \perp .

Then the element $\text{Fix } f = \bigsqcup \{f^n(\perp) \mid n \geq 0\} \in D$ exists & is the least fixpoint of f .

N.B. $f^0 = \text{id}$ & $f^n = f \cdot f^{n-1} \forall n > 0$.

Proof 7.2 - Fixpoint Theorem

First we prove that $Y = \{f^n(\perp) \mid n \geq 0\}$ is a non-empty chain.

$\perp \in Y$ trivially, since $f^0(\perp) = \perp$.

Since $f^n(\perp) \sqsubseteq f^{n+1}(\perp) \forall n \geq 0 \exists \text{Fix } f = \bigsqcup Y$ by the defining property of a chain complete partial order.

Next we show that the least upper bound of Y is the least upper bound of Y 's image under f

$$\begin{aligned} \bigsqcup Y &= \bigsqcup \{f^n(\perp) \mid n \geq 0\} \\ &= \bigsqcup \{\{f^n(\perp) \mid n > 0\} \cup \{f^0(\perp)\}\} \\ &= \bigsqcup \{\{f^n(\perp) \mid n > 0\} \cup \{\text{bot}\}\} \\ &= \bigsqcup \{f^n(\perp) \mid n > 0\} \\ &= \bigsqcup \{f \cdot f^{n-1}(\perp) \mid n > 0\} \\ &= \bigsqcup \{f \cdot f^{n'-1}(\perp) \mid n \geq 0\} \\ &= \bigsqcup \{f(y) \mid y \in Y\} \end{aligned}$$

Next, we show $\text{Fix } f$ is a fixpoint of f

$$\begin{aligned} f(\text{Fix } f) &= f(\bigsqcup Y) \\ &= \bigsqcup \{f(y) \mid y \in Y\} \\ &= \bigsqcup Y \\ &= \text{Fix } f \end{aligned}$$

Finally, we show $\text{Fix } f$ is the least fixpoint of f .

Since $f(\perp) \sqsubseteq f^n(d) \forall n > 0, d \in D$.

8 Denotational Semantics

Definition 8.1 - Denotational Semantics

A *Denotational Semantics* defines the meaning of a program using a partial function called a *state-transformer* from initial states to final states.

Denotational Semantics must be compositional (i.e. define expressions in terms of their strict sub-expressions).

Remark 8.1 - Types of Denotational Semantics

There are two types/flavours of *Denotational Semantics*

- i) Direct-Style; and,
- ii) Continuation-Style.

Definition 8.2 - Direct-Style Semantic Function

A *Direct-Style Semantic Function* has the form

$$\mathcal{S}_{ds} : \text{Stm} \rightarrow (\text{State} \hookrightarrow \text{State})$$

Proposition 8.1 - Overview of Denotational Semantics

Here is an overview of how *Denotational Semantics* decomposes common expressions.

FIX is the least fixpoint operator & cond is a conditional function.

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

Definition 8.3 - Conditional Function

The *Conditional Function* is closely related to the denotational semantics of conditionals & loops.

The *Conditional Function* is defined by

$$\begin{aligned} \text{cond} & : (X \rightarrow T) \times (X \hookrightarrow Y) \times (X \hookrightarrow Y) \rightarrow (X \hookrightarrow Y) \\ \text{cond}(b, c, d)x & = \begin{cases} c(x) & \text{if } b(x) = \text{tt} \\ d(x) & \text{otherwise} \end{cases} \end{aligned} \quad \text{Where } b \text{ is our boolean}$$

test and c & d are two functions we want to map with.

Proposition 8.2 - Conditional Function & Denotational Semantics

For *Denotational Semantics* we are interested in the case when the *Conditional Function* maps between states

$$\text{cond} : (\text{State} \rightarrow T) \times (\text{State} \hookrightarrow \text{State}) \times (\text{State} \hookrightarrow \text{State}) \rightarrow (\text{State} \hookrightarrow \text{State})$$

Definition 8.4 - Semantics of Assignment

Here is now *assignment* is defined in denotational semantics

$$\begin{aligned} \mathcal{S}_{ds}[[x := a]]s & = s[x \mapsto \mathcal{A}[[a]]s] \\ \mathcal{S}_{ds}[[x := a]] & = \lambda s. s[x \mapsto \mathcal{A}[[a]]s] \\ \mathcal{S}_{ds}[[x := a]]s \ v & = \begin{cases} \mathcal{A}[[a]]s & \text{if } v = x \\ s \ v & \text{otherwise} \end{cases} \end{aligned}$$

N.B. λ signifies lambda calculus here.

Definition 8.5 - Semantics of Skip

Here is now *skip* is defined in denotational semantics

$$\begin{aligned}
\mathcal{S}_{ds}[\text{skip}] &= \text{id} \\
\mathcal{S}_{ds}[\text{skip}] &= \lambda s.s \\
\mathcal{S}_{ds}[\text{skip}]s &= s
\end{aligned}$$

N.B. λ signifies lambda calculus here.

Definition 8.6 - Semantics of Statement Composition

Here is now *Statement Composition* is defined in denotational semantics

$$\begin{aligned}
\mathcal{S}_{ds}[[S_1; S_2]] &= \mathcal{S}_{ds}[[S_2]] \cdot \mathcal{S}_{ds}[[S_1]] \\
\mathcal{S}_{ds}[[S_1; S_2]]s &= (\mathcal{S}_{ds}[[S_2]] \cdot \mathcal{S}_{ds}[[S_1]])s \\
&= \mathcal{S}_{ds}[[S_2]](\mathcal{S}_{ds}[[S_1]]s) \\
&= s'' \text{ where } s'' = \mathcal{S}_{ds}[[S_2]]s' \text{ and } s' = \mathcal{S}_{ds}[[S_1]]s
\end{aligned}$$

N.B. $\mathcal{S}_{ds}[[S_1; S_2]]s = \text{undef}$ if $s' = \text{undef}$ or $s'' = \text{undef}$.

Definition 8.7 - Semantics of Conditionals

Here is now *Conditionals* is defined in denotational semantics

$$\begin{aligned}
\mathcal{S}_{ds}[\text{if } b \text{ then } S_1 \text{ else } S_2] &= \text{cond}(\mathcal{B}[[b]], \mathcal{S}_{ds}[[S_1]], \mathcal{S}_{ds}[[S_2]]) \\
\mathcal{S}_{ds}[\text{if } b \text{ then } S_1 \text{ else } S_2]s &= \text{cond}(\mathcal{B}[[b]], \mathcal{S}_{ds}[[S_1]], \mathcal{S}_{ds}[[S_2]])s \\
&= \begin{cases} \mathcal{S}_{ds}[[S_1]]s & \text{if } \mathcal{B}[[b]]s = tt \\ \mathcal{S}_{ds}[[S_2]]s & \text{otherwise} \end{cases}
\end{aligned}$$

N.B. $\mathcal{S}_{ds}[\text{if } b \text{ then } S_1 \text{ else } S_2]s = \text{undef}$ if the statement it calls is undefined in states s .

Definition 8.8 - Semantics of Loops

Here is now *Loops* is defined in denotational semantics

$$\begin{aligned}
\mathcal{S}_{ds}[\text{while } b \text{ do } S] &= \text{FIX } F \text{ where } F g = \text{cond}(\mathcal{B}[[b]], g \cdot \mathcal{S}_{ds}[[S]], \text{id}) \text{ Since} \\
\mathcal{S}_{ds}[\text{while } b \text{ do } S] &= \mathcal{S}_{ds}[\text{if } b \text{ then } (S; \text{while } b \text{ do } S) \text{ else skip}] \\
&= \text{cond}(\mathcal{B}[[b]], \mathcal{S}_{ds}[[S; \text{while } b \text{ do } S]], \mathcal{S}_{ds}[[\text{skip}]]) \\
&= \text{cond}(\mathcal{B}[[b]], \mathcal{S}_{ds}[[\text{while } b \text{ do } S]] \cdot \mathcal{S}_{ds}[[S]]) \\
&\in \text{fix}(\lambda g. \text{cond}(\mathcal{B}[[b]], g \cdot \mathcal{S}_{ds}[[S]], \text{id})) \\
&= \text{FIX } (\lambda g. \text{cond}(\mathcal{B}[[b]], g \cdot \mathcal{S}_{ds}[[S]], \text{id}))
\end{aligned}$$

N.B. $\mathcal{S}_{ds}[\text{while } b \text{ do } S]s = \text{undef}$ if $(\text{FIX } F)s = \text{undef}$.

N.B. λ signifies lambda calculus here.

Definition 8.9 - Functional of a Loop

The *Functional of the Loop* while b do s , when it is defined as

$$F := \lambda g. \text{cond}(\mathcal{B}[[b]], g \cdot \mathcal{S}_{ds}[[S]], \text{id})$$

The functional can be seen as a means of finding better approximations to the semantics of the loop as it can be shown that

$F^n(\emptyset)$ is a correct semantics for all states from which the loop ends in fewer than n iterations.

N.B. λ signifies lambda calculus here.

Proposition 8.3 - Characterisation of the Functional of a Loop

A *direct characterisation* of F is any equivalent mathematical expression that does not contain any semantic functions.

Remark 8.2 - Definition of Loop is insufficient

The fixpoint functional definition for the semantics of loops is not sufficient

$$\mathcal{S}_{ds}[\text{while } b \text{ do } S] = \text{FIX } F \text{ where } F g = \text{cond}(\mathcal{B}[[b]], g \cdot \mathcal{S}_{ds}[[S]], \text{id})$$

It faces two problems

- i) There are functionals that have more than one fixed point;
- ii) There are functionals that have no fixed points.

Proposition 8.4 - Defining Semantics for Loops using Fixpoints

The following example shows that fixpoints of a loop functional are not sufficient, in themselves, to provide a semantics for loops as more than one such fixpoint may exist.

Consider the loop `while $\neg(x = 0)$ do skip`. The function F of this is defined as

$$F\ g = \text{cond}(\mathcal{B}[\neg(x = 0)],\ g \cdot \mathcal{S}_{ds}[\text{skip}],\ \text{id})$$

$$\text{Thus } F\ g\ s = \begin{cases} g\ s & \text{if } s\ x \neq 0 \\ s & \text{if } s\ x = 0 \end{cases}.$$

From this we can identify the fixpoints s where

$$g\ s = F\ g\ s = \begin{cases} g\ s & \text{if } s\ x \neq 0 \\ s & \text{if } s\ x = 0 \end{cases}$$

But since $g\ s == g\ s$ is always true the top condition is redundant and so a fixpoint of F is just any function g st $g\ s = s$ i.e. where $s\ x = 0$.

Thus the least fixpoint is undefined whenever $s\ x \neq 0$, even though other fixpoints exist (e.g. `id`).

Remark 8.3 - Functionals with no fixpoints

Let g_1 & g_2 be any state transformers st $g_1 \neq g_2$.

Define the functional F as

$$F\ g = \begin{cases} g_1 & \text{if } g = g_2 \\ g_2 & \text{otherwise} \end{cases}$$

Remark 8.4 - Motivation to Solution to Remark 7.3

Consider the three possible outcomes for a state s_0 in a loop

- i) Loop Terminates
e.g. The loop `while $0 \leq x$ do $x := x - 1$` terminates $\forall s_0$ where $x \geq 0$.
- ii) Loops Locally, loops due to a construct with statement S .
e.g. The loop `while $0 \leq x$ do (if $x = 0$ then (while true do skip) else $x := x - 1$)` ends up looping once $x = 0 \forall s_0$ where $x \geq 0$.
- iii) Loops Globally, loops due to condition b .
e.g. The loop `while $\neg(x = 0)$ do skip` loops $\forall s_0$ where $x \neq 0$.

9 Static Program Analysis

THIS CHAPTER IS NON-EXAMINABLE.

10 Axiomatic Semantics

Remark 10.1 - Motivation

Generally we are more interested in some properties of a program, than others.

The more interesting properties tend to be those related to the intended purpose of the program, rather than incidental results due to a particular implementation.

e.g. Given `while $\neg(x=1)$ do (u:=y*x; x:=x-1)` we are interested that $y = n!$ when initially

$y = 1$ & $n > 0$, but not interested in the final value of x .

Definition 10.1 - Partial Correctness Properties

Partial Correctness Properties state that if a program terminates then a certain relationship will hold between the initial & final variable values.

Definition 10.2 - Total Correctness Properties

Total Correctness Properties state that a program will terminate and a certain relationship will hold between the initial and final variables values.

Remark 10.2 - Axiomatic Semantics & Correctness Properties

An *Axiomatic Semantics* allows us to prove a program satisfies formal partial or total correctness properties.

Definition 10.3 - Axiomatic Semantics

An *Axiomatic Semantics* consists of a set of axioms/rules that provide a method for finding true assertions.

An *Axiomatic Semantics* formalises interesting properties of a program by defining pre-conditions & post-conditions.

Remark 10.3 - Transitions of Axiomatic Semantics

An *Axiomatic Semantics* will have a single assertion a , the conclusion, and one or more assertions a_1, \dots, a_n , premises, which are required to prove a holds

$$\frac{a_1 \dots a_n}{a}$$

Definition 10.4 - Hoare Triples

Hoare Triples are the standard way of denoting axiomatic semantics

Pre-Condition Program Post-Condition

N.B. The conditions are known as assertions.

Remark 10.4 - Partial Correctness Semantics & Hoare Triples

Under the *Partial Correctness Semantics* assertions are written as $\{P\} S \{R\}$.

This means that if P holds before running S and S terminates, then R will hold immediately after.

Remark 10.5 - Total Correctness Semantics & Hoare Triples

Under the *Total Correctness Semantics* assertions are written as $\{P\} S \Downarrow R$ or $[P] S [R]$.

This means that if P holds before running S then R will hold immediately after.

Definition 10.5 - Logical Variables

Logical Variables are variables that are not defined in the running of a program, but can be derived from program variables.

N.B. Commonly *Logical Variables* are used to remember past values of program variables.

Definition 10.6 - Schemata

Schemata are used to define individual axioms & contain meta-variables that may be instantiated for particular pre & post-conditions.

Proposition 10.1 - *Intentional Partial Correctness Schemata*

$\{P\} \text{ skip } \{P\}$	Skip
$\{P(a)\} x := a \{P(x)\}$	Assertion
$\frac{\{P\} S_1 \{R\} \quad \{R\} S_2 \{Q\}}{\{P\} S_1; S_2 \{Q\}}$	Composition
$\frac{\frac{\{P\} S_1; S_2 \{Q\}}{\{P \wedge b\} S_1 \{Q\}} \quad \{P \wedge \neg b\} S_2 \{Q\}}{\{P\} \text{ if } b \text{ then } S_1 \text{ else } S_2 \{Q\}}$	Conditional
$\frac{\{P\} \text{ if } b \text{ then } S_1 \text{ else } S_2 \{Q\}}{\{P \wedge b\} S \{P\}}$	Loop
$\frac{\{P\} \text{ while } b \text{ do } S \{P \wedge \neg b\}}{\{P'\} S \{Q'\} \text{ if } P \models P' \ \& \ Q' \models QP}$	Consequence
$\{P\} S \{Q\}$	

N.B. $P \models Q \implies$ if P is true then Q is true.

Proposition 10.2 - *Extensional Partial Correctness Schemata*

$\{P\} \text{ skip } \{P\}$	Skip
$\{P[x \mapsto A_{[[a]]}]\} x := a \{P\}$	Assertion
$\frac{\{P\} S_1 \{R\} \quad \{R\} S_2 \{Q\}}{\{P\} S_1; S_2 \{Q\}}$	Composition
$\frac{\frac{\{P\} S_1; S_2 \{Q\}}{\{P \wedge B + [[b]]\} S_1 \{Q\}} \quad \{P \wedge \neg B_{[[b]]}\} S_2 \{Q\}}{\{P\} \text{ if } b \text{ then } S_1 \text{ else } S_2 \{Q\}}$	Conditional
$\frac{\{P\} \text{ if } b \text{ then } S_1 \text{ else } S_2 \{Q\}}{\{P \wedge B_{[[b]]}\} S \{P\}}$	Loop
$\frac{\{P\} \text{ while } b \text{ do } S \{P \wedge \neg B_{[[b]]}\}}{\{P'\} S \{Q'\} \text{ if } P \implies P' \ \& \ Q' \implies Q}$	Consequence
$\{P\} S \{Q\}$	

Remark 10.6 - *Using Consequence Rule*

One should avoid using the *Consequence Rule* unless no other rules would be valid.

When used you need to also prove that the new conditions ($P' \ \& \ Q'$) do satisfy $P \models P' \ \& \ Q' \models Q$.

Example 10.1 - *Modulus*

$$\frac{\frac{\frac{\{(x-2)\%2 = n\%2 \ \&\& \ x-2 \geq 0\} x := x-2 \{x\%2 = n\%2 \ \&\& \ x \geq 0\}}{\{x\%2 = n\%2 \ \&\& \ x \geq 0 \ \&\& \ 2 \leq x\} \text{ while } 2 \leq x \text{ do } x := x-2 \{x\%2 = n\%2 \ \&\& \ x \geq 0\}}}{\{x\%2 = n\%2 \ \&\& \ x \geq 0\} \text{ while } 2 \leq x \text{ do } x := x-2 \{x\%2 = n\%2 \ \&\& \ x \geq 0 \ \&\& \ \neg(2 \leq x)\}}}{\{x = n \ \&\& \ n \geq 0\} \text{ while } 2 \leq x \text{ do } x := x-2 \{x = n\%2\}}$$

Proposition 10.3 - *Total Correctness Schemata*

$[P] \text{ skip } [P]$	Skip
$[P(a)] x := a [P(x)]$	Assignment
$\frac{[P] S_1 [Q] \quad [Q] S_2 [R]}{[P] S_1; S_2 [R]}$	Composition
$\frac{\frac{[P] S_1; S_2 [R]}{[P \wedge b] S_1 [Q]} \quad [P \wedge \neg b] S_2 [Q]}{[P] \text{ if } b \text{ then } S_1 \text{ else } S_2 [Q]}$	Conditional
$\frac{[P] \text{ if } b \text{ then } S_1 \text{ else } S_2 [Q]}{[P(z+1)] S [P(z)]} \text{ if } \models \forall z \in \mathbb{N} P(z+1) \implies b$	Loop
$\frac{[\exists z \in \mathbb{N} P(z)] \text{ while } b \text{ do } S [P(0)]}{[P'] S [Q']} \text{ if } \models P(0) \implies \neg b$	
$\frac{[P'] S [Q']}{[P] S [Q]} \text{ if } P \models P' \ \& \ Q' \models Q$	Consequence

0 Reference

0.1 The While Language

Definition 0.1 - *While*

While is a simple Turing-Complete language which has the following *syntactic categories*

Numerals	0,1,2,...
Variables	x,y,temp,...
Arithmetics	+,*,-
Booleans	true,false,=,<=,¬,∧
Statements	:=,;, skip, if...then...else, while...do

Definition 0.2 - *Meta-Variables*

Meta-Variables are the notation used to denote the category of a generic variable.

Subscripts & priming notation is used to distinguish between variables of the same category.

The following are the *meta-variables* for each category

Num	n
Var	x
Aexp	a
Bexp	b
Statement	S

Definition 0.3 - *Basis Element*

A *boolean expression* is said to be a basis element if it is **true**, **false** or has the form $a_1 = a_2$ or $a_1 \leq a_2$.

N.B. Either is a standard value or compares arithmetic expressions.

Definition 0.4 - *Composite Element*

A *boolean expression* is said to be a *Composite Element* if it has the form $b_1 \wedge b_2$ or $\neg b$.

N.B. It compares boolean expressions.

Definition 0.5 - *Abstract Syntax of While*

There are three types of abstract syntax in the *While* language: Arithmetic Expressions, Boolean Expressions; & Statements

Aexp	::=	Num—Var
		Aexp '+' Aexp
		Aexp '*' Aexp
		Aexp '-' Aexp
Bexp	::=	'true' 'false'
		Aexp '=' Aexp
		Aexp '<=' Aexp
		'¬' Bexp
		Bexp '∧' Bexp
Stm	::=	Var ':=' Aexp
		'skip'
		Stm ';' Stm Aexp
		'if' Bexp 'then' Stm 'else; Stm
		'while' Bexp 'do' Stm

Definition 0.6 - *Concrete Syntax of While*

The follow are concrete syntaxes of *While*

Digit ::= '0'|'1'|\dots|'9'
 Letter ::= 'a'|'b'|\dots|'z'
 String ::= ϵ |Digit String |Letter String
 Num ::= Digit|Digit Num
 Var ::= Letter String
 Id ::= Num|Var
 Factor ::= Id|Factor '*' Id
 Aexp ::= Factor |Aexp '+' Factor |Aexp '-' Factor
 Atom ::= 'true'|'false'|Aexp '=' Aexp|Aexp '<=' Aexp
 Literal ::= Atom|'¬' Literal
 Bexp ::= Literal|Bexp '^' Literal

Proposition 0.1 - *Update Notation*

$f[x \mapsto y]$ is a function that updates the function f to return y for value x

$$(f[x \mapsto y])x' = \begin{cases} y & \text{if } x' = x \\ f \ x' & \text{otherwise} \end{cases}$$

Definition 0.7 - *Arithmetic Semantics*

Here is a definition for the function \mathcal{A} which returns the value of arithmetic expressions

$$\begin{aligned}
 \mathcal{A} &: \text{Aexp} \rightarrow (\text{State} \rightarrow \mathbb{Z}) \\
 \mathcal{A}[[n]]s &= \mathcal{N}[[n]] \\
 \mathcal{A}[[x]]s &= \text{sx} \\
 \mathcal{A}[[a_1 + a_2]]s &= \mathcal{A}[[a_1]]s + \mathcal{A}[[a_2]]s \\
 \mathcal{A}[[a_1 * a_2]]s &= \mathcal{A}[[a_1]]s * \mathcal{A}[[a_2]]s \\
 \mathcal{A}[[a_1 - a_2]]s &= \mathcal{A}[[a_1]]s - \mathcal{A}[[a_2]]s
 \end{aligned}$$

Definition 0.8 - *Boolean Semantics*

Here is a definition for the function \mathcal{B} which returns the boolean value of an expression.

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

Definition 0.9 - *Binary Number Semantics*

Here is a definition for the function \mathcal{N} which returns the denary value of binary number

$$\begin{aligned}
 \mathcal{N} &: \text{Num} \rightarrow \mathbb{Z} \\
 \mathcal{N}[[0]] &= 0 \\
 \mathcal{N}[[1]] &= 1 \\
 \mathcal{N}[[n \ 0]] &= 2 * \mathcal{N}[[n]] \\
 \mathcal{N}[[n \ 1]] &= 1 + 2 * \mathcal{N}[[n]]
 \end{aligned}$$