Program Language Design

Lecturer:
Rosemary Monahan
Room: 2.105 Callan Building

Email: Rosemary.Monahan@may.ie

Course Text

Denotational Semantics: A Methodology for Language Development* by *David Schmidt, Kansas State University

Full text is available at:
http://www.cis.ksu.edu/~schmidt/text/densem.html

Other supporting texts:
- Types and Programming Languages by B.J. Pierce, MIT Press
- A practical introduction to Denotational Semantics: Lloyd Allison, Cambridge Press

Course Assessment

Total marks: 1000

Examination: 1 x 2 hour written exam at the end of semester 1: 800 marks
Continuous Assessment: 200 marks

Penalties: (*for late submission of Course/Project Work etc.)*
Late continuous assessment will result in a penalty of 20% reduction per day in the amount of marks attainable for the practical element of the course.

Programming languages

- Many different languages
- Programmer must fully master the languages which they use to solve problems
- They must know what phrases are permitted
- They must know what these phrases mean
- Language descriptions are usually in the form of:
  - Program Language reference manuals
  - Programming guides
  - **Program Language Definition- plays a central role as all other descriptions are based on it. It must give a complete and precise account of the language as possible …**

Programming languages ...

- Many different languages
- Programmer must fully master the languages which they use to solve problems
- They must know what phrases are permitted
- They must know what these phrases mean
- Language descriptions are usually in the form of:
  - Program language reference manuals
  - Programming guides
  - **Program Language Definition- plays a central role as all other descriptions are based on it. It must give a complete and precise account of the language as possible …A mathematical based description of the language.**

Characteristics of Programming Languages

- Syntax: the appearance and structure of a languages sentences - determines which symbol sequences are permitted phrases of the language
- Semantics: the assignment of semantics / meanings to the sentences of a programming language. Explains what the various phrases of a language mean/denote.
- Pragmatics: the usability of a language, including the possible areas of application of the language, its ease of implementation and use, and the languages success in fulfilling its stated goals.

Program Language Design

- **Denotational Semantics** is a method for defining the semantics of programming languages.
- It is of interest to the language designer, compiler writer and programmer.
- It should be concise, unambiguous, open to mathematical analysis, mechanically checkable, executable and readable … depending on your point of view.
- Denotational semantics is based on well understood mathematical foundations and uses a well defined notation/ meta-language…

Syntax:

- The area of syntax specification has been thoroughly studied and Backus Naur Form (BNF) is widely used for defining syntax.

  <loop>::= while <Boolean expression> do <compound statement>

  The underlying formalism are context free grammars which in turn are based on set theory.
- Note: correspondence between languages BNF definition and parsers.

Semantics

Semantic definition methods provide:

- A precise standard for a computer implementation guaranteeing that the language is implemented exactly the same on all machines and enhancing portability.
- Useful user documentation
- A tool for design and analysis
- Input to a compiler which maps a semantics definition to a guaranteed correct implementation for the language.
- A mathematical basis for program verification

Interpreted Code:
The semantics of the programming language is defined entirely by the interpreter

- Parser: reads the input and ensures that it has the correct syntax
- Evaluation Module: Evaluates input into to its corresponding output and hence defines the input semantics.

Compiled Code:
The Compiler transforms the input program into fast running machine code – preserving the meaning of the original program. The semantics of the program is independent of the compiler or computer.

Methods for Semantics Specification

1. Operational Semantics:

- the meaning of a program in the language is the evaluation history that the interpreter produces when it interprets the program (the behaviour is defined in terms of the behaviour or an abstract machine which operates according to the structure of a program)
- Problems: No machine independent definition exists & we need to write the interpreter!
- Advantage: Implement the interpreter and we have implemented the language

1.Operational Semantics

- In operational semantics we are concerned with the change of the machines state when a phrase is executed.
- Example:
  Computation : Statements x State ⟶ State*
  Computation (u = e, s)
  = s'.(update (z,u,Store)), Input', Output', Environment')

1. Operational Semantics

- In operational semantics we are concerned with the change of the machines state when a phrase is executed.
- Example:
  Computation : Statements x State ⟶ State*
  Computation (u = e, s)
  = s'.(update (z,u,Store)), Input', Output', Environment')
  where a State consists of a Store, an Input stream, an Output Stream and an Environment, dashed variables represent new states and z is the result of evaluating e in state s.

2. Axiomatic Semantics

- The meaning of the programming language is defined implicitly by a logical calculus called program logic which provides a tool for the derivation of assertions of the form:
  {Precondition} Program {Postcondition}
- Properties about program language constructs are defined and expressed with axioms and rules from logic. A property about a program is deduced by using the axioms and rules to construct a formal proof of the property.
- Axiomatic definitions tend to be abstract and are best used at the specification stage or to give documentation of language properties which are of interest to the user. Also used in algorithm derivation.

2. Axiomatic Semantics

- Example:
  Assignment:

Precondition:	{ x>= 6 }
Program:	x = x + 7
Postcondition:	{x>=13}

3. Denotational Semantics

- This method maps a program directly to its meaning called its denotation. The denotation is usually a mathematical value, such as a number or a function.
- No interpreters are used, a valuation function maps a program directly to its meaning.
- A denotational definition is more abstract than an operational definition, as it does not specify computation steps.
- Its high level, modular structure makes it especially useful to language designers and users, as the individual parts of a language can be studied without having to examine the entire definition.

3. Denotational Semantics

- Denotational semantics views a **function** as a probable set of ordered input/output pairs and an **algorithm** as a finite description of the function.
- A **program** is an algorithm written in some particular programming language. A program stands for, or denotes, a function.
- A denotational semantics of a programming language gives the mapping from programs in the language to the functions denoted.

Example: Factorial = { <0,1>, <1,1>, <2,2>, <3,6>, …}
fac(n) = if n = 0 then 1 else n * fact(n-1)
DS confirms that program fac denotes the factorial function.

Denotational Semantics Example

- Decimal numbers form a language, **NUM,** over the alphabet {0,1,2,3,4,5,6,7,8,9} which can be defined by the grammar

v ::= vδ | δ

δ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

- The decimal numbers are usually taken to denote, integers which are abstract objects. This interpretation can be made formal by giving a valuation function **V**:

V: NUM -> Int

where **V** is a function from the sentences in the language **NUM** to the integers **Int**.

Denotational Semantics Example:

V :Num -> Int
V[vd] = (10 * V[v])+ V[d]

V[0] = 0, V[1] = 1, V[2] = 2, V[3] = 3
V[4] = 4, V[5] = 5, V[6] = 6, V[7] = 7
V[8] = 8, V[9] = 9

Denotational Semantics Example

- **V: NUM -> Int**
- **V[123] = (10 * V[12]) + V[3]**
  = (10 * V[12])+ 3
  = (10 * (10 * V[1] + V[2])) + 3
  = (10 * (10 * 1 + 2)) + 3
  = (10 * 12) + 3
  = 120 + 3
  = 123

Designing a new language

- Designers supply a list of properties that they wish to system to have
- An axiomatic semantics is given defining the input language and how it achieves the desired properties
- A denotational semantics is then defined to give a meaning for the language
- A formal proof is constructed to show that the semantics contains the properties that the axiomatic semantics specifies
- The denotational semantics is implemented using an operational definition.
- Thus, these complimentary semantic definitions of a language support systematic design, development and implementation.

Section 1: Syntax

Syntax

- Concrete Syntax defines the way the language is actually written on paper or on a screen and includes sufficient information to parse it.
- Abstract Syntax specifies the relations between logical parts of a the language, it can be simpler and may not contain enough information to parse the language unambiguously.
- Example:
  y ::= if ε then y else y fi concrete
  y ::= ε → y , y abstract

Syntax

Concrete:
ε ::= ε + τ | τ
τ ::= τ * Ψ| Ψ
Ψ::= (ε) | a | b | c |...
Note: This specifies that a + (b*c) should be parsed as a+(b*c)
The sequence of the rules gives the operators = , *, (), in increasing order of priority.

Abstract: loses this information
ε ::= ε + ε | ε * ε | a | b | c| …

Syntax

- Symbols for building words
- Structure of words
- Structure of well formed phrases
- Structure of sentences

Only syntactically correct programs also have a semantics

	Arithmetic:	:
Symbols:	0-9, + , -, *, ?, (,)	
Words:	Numerals built from symbols and the operators	
Phrases:	Arithmetic expressions	
Sentences:	Phrases	

Prog. Language

Symbols:	letters, digits, operators
Words:	identifiers, numerals, operators
Phrases:	expressions, statements
Sentences:	programs

BNF

Specification of formal languages

- Sets of equations
- left-hand-side: non terminal
  - name of a structured type
- right hand side: list of forms
  - terminal(symbols) and non-terminals
  - <non-terminal> ::= form1 | form2 | …| form n

Syntax

Example:

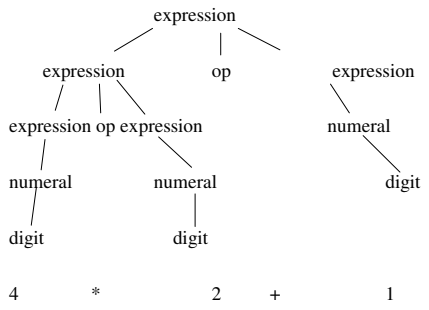
<digit> ::= 0 | 1 | 2 | 3| 4 | 5 | 6 | 7 | 8 | 9 |

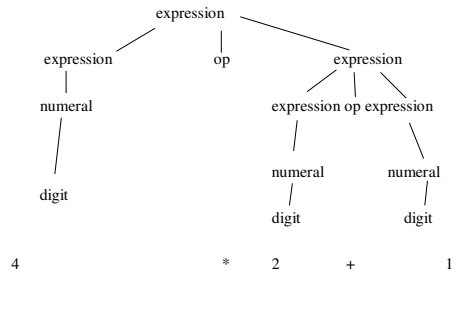
<operator> ::= + | - | * | /

<numeral> ::= <digit> | <digit><numeral>

<expression> ::= <numeral> | (<expression> | <expression> <operator> <expression>

The structure of an expression is illustrated by its derivation tree





Unambiguous syntax definitions

- Expression 4*2+1 has two derivation trees
- Add syntax definitions: lose the ambiguity

<expression> ::= <expression><lowop><term> | <term>
<term> ::= <term> <highop><factor> | <factor>
<factor> ::= <numeral> | (<expression>)
<lowop> ::= + | -
<highop> ::= * | /

Extra level of structure makes derivation unique but syntax complicated with artificial levels of structure.

Semantics

- We do not need to use artificially complex BNF definitions!
- Why?
- Derivation trees are the real sentences of the language.(Strings of symbols are just abbreviations of trees, these abbreviations may be ambiguous)

In real life we use two BNF Definitions:

- Abstract Syntax: Analyse structure of tree and determine its semantics
- Concrete Syntax which is derived from the abstract -determine derivation tree from string abbreviation (parsing) introducing non interesting problems

Concrete definitions handle parsing problems but the abstract definition is the true definition of the language structure.

Abstract Syntax Definitions

- Abstract Syntax Definitions are descriptions of structure
- Terminal symbols disappear - meaning assigned to words, not symbols
- Building blocks are words

Abstract syntax is studied at the word level.

<expression > ::= <numeral>| <expression><operator><expression>|
left-paren<expression> right-paren

<operator>::=plus | minus | mult | div

<numeral> ::= zero | one | … | ninety| …

Structure remains, text vanishes, tree's leaves are tokens instead of symbols.

Set Theory

More abstract view of abstract syntax

- non-terminal names set of phrases specified by corresponding BNF rule - Expression, Op, Numeral
- Rules replaced by syntax builder operations, one for each form of rule.
  - Numeral-exp: Numeral -> Expression
  - Compound-exp:
    - Expression x Op x Expression -> Expression
  - Bracket-exp:Expression -> Expression
- Terminal words replaced by constants
  - plus:Op
  - zero: Numeral …

Words and derivation trees replaced by sets and operations.

More readable version

- Syntax domains
- BNF rules

Abstract Syntax

E ∈ Expression
O ∈ Operator
N ∈ Numeral

E ::= N | EOE | (E)
O ::= + | - | * | /
N is a set of values

A block structured programming language

Abstract Syntax

P ∈ Expression, B ∈ Block, D ∈ Declaration, C ∈Command,
E ∈ Expression, O ∈Operator, I ∈ Identifier, N ∈ Numeral.

P ::= B
B ::= D,C
D ::= var I | procedure I, C | D1;D2
C ::= I := E | if E then C | while E do C | C1;C2 | begin B end
E ::= I | N | E1 O E2 | (E)
O ::= + | - | * | div

Interactive file editor

Abstract Syntax
P ∈ Program session, S ∈ Command Sequence, C ∈ Command,
R ∈ Record, I ∈ Identifier

P ::= S cr
S ::= C cr S | quit
C ::= newfile | open I | moveup| moveback | insert R | delete | close

Mathematical Induction

- Strategy for proving P on natural numbers.
  - Induction basis: Show that P(0) holds
  - Induction Hypothesis: assume P(i)
  - Induction Step: prove P(i+1)

Proposition:
The following equation is true for all natural numbers:

$$2 + 4 + 6 + 2n = n(n+1)$$

Proof: By Induction – Try it!

Structural Induction

- Mathematical induction relies on the structure of the natural numbers:
  N ::= 0 | N+1
  - Show that all trees of zero depth has property P
  - Assume trees of depth m or less have property P
  - Prove that the tree of depth m + 1 has property P
- Arbitrary syntax domains:
  D ::= Option1 | Option2 | Option3 | …| Option n
  To prove that all members of D have P
  - assume occurences of D in option i have P
  - prove that option i has P (for each option i)

Structural Induction-Example 1:

For the domain E:Expression and its BNF rule:
E ::= zero | E1*E2 | (E) show that all members of Expression have the same number of left and right parentheses.
Proof:Consider the three options:

- zero- there are zero occurrences of both left and right
- E1 * E2: by the inductive hypothesis, E1 has say m left parentheses and m right parentheses, and similarly E2 has n left and n right parenthesis. Then E1*E2 has m + n left and m + n right parentheses.
- (E): by the inductive hypothesis, if E has m left and m right parentheses then (E) has m+1 left and m+1 right parentheses

Structural Induction- Example 2:

- The Structural induction principle generalises to operate over a number of domains simultaneously. We can prove properties of two or more domains that are defined in terms of one another.

- For BNF rules:
  S ::= *E*
  E ::= +S | **
  show that all S-values have an even number of occurrences of the* token.

Example 2:

Proof:
S and E are mutually recursively defined hence the result must be proved by a simultaneous induction on their rules.
For Rule S:The only option is that by the induction hypothesis, the E tree has an even number of *, say m of them.
Then the *E* tree has M + 2 of the, which is an even value.
For Rule E:The 1st option builds a tree that has an even number of *, because by the inductive hypothesis, the S tree has an even number, and no new ones are added. The second option has exactly two occurences, which is an even number.

Section 2: Sets, Functions, Domains

Introduce functions through set theory and examines the concepts of set theory which forms a foundation for the theory of semantic domains.

Sets

Collection of elements

- Enumeration of elements
  {1,{1,4,7}, 4}, {red, yellow}, { } (also written as ∅)
- Defining Property
  {x | P(x)}
  {x | x is an even integer}

Examples:

Natural numbers	N = {0,1,2,3,4,5,…}
Integers	Z = { …-2,-1, 0, 1, 2, …}
Truth Values (Booleans)	B = {True, False}
Rational Numbers	Q = {x|x = p/q for some p,q ∈ Z, q ≠0}

Set Predicates

Based on the concept of membership

- Membership: x ∈ S
  - only basic predicate - sets are black boxes otherwise
- Equivalence: R = S
  - x ∈ R ⇔ x ∈ S (for all x)
  - extensionality principle
    { 1, 4, 7} = {4, 1, 7}, {1,4,7} = {4, 1, 7, 4}
- Subset: R ⊆ S
  - x ∈ R ⇒ x ∈ S (for all x)
  - ({ } ⊆ S, S ⊆ S)

Set Constructions

Building new sets

- Union: R ∪ S
  {x|x ∈ R or x ∈ S}
  Commutative: R ∪ S = S ∪ R
  Associative: (R ∪ S) ∪ T = R ∪ (S ∪ T)
  Intersection: R ∩ S
  {x|x ∈ R and x ∈ S}
  Also Commutative and Associative
- Powerset: **P**(R)
  {x|x ⊆ R}
  ({ } ∈ P(R), R ∈ **P**(R))

Pairs & Products

Concept of ordered pair

- Constructor (x,y)
- Selectors
  - fst,
  - snd
- Pair Equivalence
- For sets R and S, their Product, R **x** S is the set of all pairs built from R and S. R **x** S = {(x,y) | x ∈ R and x ∈ S}
- Both pairing and products can be generalised from their binary formats to n-tuples and n-products.

Sum

A form of union construction on sets that keeps the members of the respective sets R and S separate is called disjoint union (or sum):

R + S= {(zero, x) | x} ∪ {(one,y)| y ∈ S}
"tags" to preserve origin of element

- Constructors
  inR(x) = (zero,x) (for x ∈ R)
  inS(x) = (one,y) (for y ∈ S)

Sum

- Selectors
- To remove a tag from an element m ∈ R+ S, we use cases. For any m ∈ R+ S the value of

```
cases m of
    isR(x) …x…
    [] isS(y) … y …
end
```

is …x… when m = (zero,x) and …y… when m = (one,y)

Functions

- Black box that accepts objects as input and produces another object as output
- Definition in terms of sets
- $f:R \rightarrow S$
  - f is a function from R to S
  - R: domain of f, S: co-domain of f
  - $R \rightarrow S$ arity (functionality) of f
- Application f(a)
  - $a \in R$, $f(a) \in S$
- Equality f = g
  - $f,g:R \rightarrow S$ f(x) = g(x) (for all x)

- Composition f o g (Associative)
  - $f:R \rightarrow S$, $g:S \rightarrow T$
  - $g \circ f :R \rightarrow T$ //check notes
  - $(g \circ f)(x) = g(f(x))$
- Mappings:
  - Injective (1-1) $f(x) = f(y) \Rightarrow x = y$
  - Surjective (onto) $\forall y \in S, \exists x \in R: f(x) = y$
  - Identity $f:R \rightarrow R, f(x) = x$ (for all $x \in R$)
  - Inverse $f:R \rightarrow S$ injective and surjective
    $g:S \rightarrow R, g(y) = x \Leftrightarrow f(x) = y$
    $g = f^{-1}$

Isomorphism

- Relationship between sets defined by functions
- R and S are isomorphic if there is a pair of functions
  - $f : R \rightarrow S$
  - $g : S \rightarrow R$
  - f o g is identity on R , g o f is identity on S
  - A fct is an isomorphism iff it is 1:1 and onto.
- f and g are then called isomorphism's.

Examples :

- R = {1, 4, 7} is isomorphic to S = {2, 4, 6}
- A x B is isomorphic to B x A; take $f:AxB \rightarrow BxA$ to be f(a,b) = (b,a)

- **Example:**
  - N is isomorphic to Z; Take $f: N \rightarrow Z$ to be
  - f(x) = x/2 if x is even, - ((x+1) /2) if x is odd

Functions as Sets

- Every function $f : R \rightarrow S$ can be represented by its graph:
  - $graph(f) = \{(x, f(x)) \mid x \in R\} \subseteq R \times S$
- Successor function on Z {…,(-2,-1), (-1,0), (0,1), (1,2), …}
- Function application
  - $f(a) = b \Leftrightarrow (a,b) \in graph\ f$
  - f(a) = apply(graph(f), a)
- Function composition
  - graph(g o f) =
    {(x, z)| x ∈ R and, for some y ∈ S,
    (x, y) ∈ graph(f) and (y, z) ∈ graph(g)}

Examples:

- **add : (N x N) → N**
  graph(add) = {((0, 0), 0), ((1, 0), 1), ((0, 1), 1), ((1, 1), 2), ((2, 0), 2), ((2, 1), 3), ((2, 2), 4), …}
- **duplicate : R → R x R, where R = {1, 4, 7}**
  graph (duplicate) = {(1, (1, 1)), (4, (4, 4)), (7, (7, 7))}
- **which : (B +N) →S where S = { isbool, isnum}**
  graph (which) = {((zero, true), isbool), ((zero, false), isbool), ((one, 0), isnum), ((one, 1), isnum), ((one, 2), isnum), …}
- **singleton : N → P(N)**
  graph(singleton) = {(0, {0}), (1, {1}), : : : , (n, {n}), …g}
- **nothing : B ∩ N → B**
  graph (nothing) = { }

- Graphs make it clear how the function behave when they are applied to arguments e.g.
  - Apply(graph(which), (one, 2)) = num
- Since a function can be represented by its graph, which is a set we will allow functions to accept other functions as arguments and produce functions as answers.
- A function that uses functions as arguments or results is called a **higher order function**. Their graphs become complex rather quickly, but they do exist and are legal under the laws of set theory.

Functions as Equations

- The graph representation of a function provides insight into its structure but is inconvenient to use in practice.
- add : (N x N) → N
  - add(m, n) = m+ n
- duplicate : R → R x R, = {1, 4, 7}
  - duplicate(r) = (r, r)
- which : (B +N) → {isbool, isnum}
  which(m) = cases m of isB(b) → isbool; isN(n) → isnum end
- singleton : N → P(N)
  - singleton(n) = {n}
- nothing : B ∩ N → B
  - no equational definition (domain empty)!

Evaluation of Equations

- Definition f : A → B, f(x) = α
- Application f(a_0)
  - 1. Substitution [a_0 /x] α
  - 2. Simplification to underlying value
  - E.G. Add(2,3)= [3/n][2/ m] m + n = 2 + 3 = 5
- Lambda Notation f = λx: α
  - add(x, y) may be defined as λ (xy).x + y
  - [3/x][2/ y] x + y = 2 + 3 = 5
- Updating Functions [$a_0 \rightarrow b_0$]f
  - ([$a_0 \rightarrow b_0$] f)(a_0) = b_0
  - ([$a_0 \rightarrow b_0$]f)(a) = f(a), for all $a \neq a_0$

Semantic Domains

- Those sets that are used as value spaces in denotational semantics are called semantic domains. To build a domain we make use of
  - Primitive domains N, Z, B, ...
  - Compound domains
    - Product domains A x B
    - Sum domains A+ B
    - Function domains A → B
    - Lifted domains $A_\perp = A \cup \{\perp\}$ (alternative to partial fcts)
      - ⊥ = ``bottom" , Non-termination, no value at all
      - Strict functions f : $A_\perp \rightarrow B_\perp$, f = λx. α denotes
        - » f(⊥) = ⊥
        - » f(a) = [a/x] α , for a ∈ A

Semantic Algebras

- Format for presenting semantic domains
  - clearly states the structure of a domain and how its elements are used by the functions
  - Encourages the development of a standard algebra module that may be used with many semantic definitions
  - Makes it easier to analyse a semantic definition concept by concept.
  - Makes it straightforward to alter a semantic definition by replacing one semantic algebra with another

Example of a Semantic Algebra

- Rational Numbers - Domain Rat = (Z x Z) $_{\perp}$
- Operations
  - makerat : Z → (Z → Rat)
  - makerat = λ p.λ q.(q = 0) → ⊥ [] (p, q)
  - addrat : Rat → (Rat → Rat)
  - addrat = λ (p_1 , q_1): λ (p_2 , q_2):((p_1 * q_2)+(p_2 * q_1), q_1 * q_2)
  - multrat : Rat → (Rat → Rat)
  - multrat = λ (p_1 , q_1): λ (p_2 , q_2):(p_1 * p_2 , q_1 * q_2)
- Notes:
  1. Choice function $e_1 \rightarrow e_2$ [] e_3 $\Rightarrow e_2$, if e_1 = true, $\Rightarrow e_3$, if e_1 = false
  2. (p,q) represents p/q

Section 3: Semantic Algebras

Describes semantic domains, their associated construction, destruction, and its presentation in a semantic algebra format.

Primitive domains e.g. Nat, Bool, String
Compound domains e.g. Product, Function Space, Recursive Domains

Domain Theory

- Domain Theory: The study of "structured sets" and their operations.
- Fundamental concept: semantic domain- a set of elements grouped together because they share some common property or use. The set of natural numbers is a useful semantic domain, its elements are structurally similar and share common use in arithmetic.
- Domains may be nothing more than sets but there are situations in which other structures such as lattices or topologies may be used instead. For the moment we assume all domains are simply sets.

Operations

- A set of operations accompany domains. The operations are functions that need arguments from the domain to produce answers.
- Operations are defined in two parts:
  - the operations domain and codomain are given by the operations functionality. For an operation f, its functionality f: D_1 x D_2 x .. x $D_n \rightarrow$A says that f needs an argument from domain D_1 and one from D_2 to produce an answer in domain A.
  - A description of the operations mapping is specified. This is usually an equational definition, but a set graph, table or diagram may also be used.

A Domain plus its operations constitutes a **semantic algebra**.

Primitive Domains

- Primitive Domain: a set that is fundamental to the application been studied. Its elements are atomic and they are used as answers or semantic outputs e.g. The real number are a primitive domain for mathematicians.
- Example:Natural Numbers

  Domain Nat = N

  Operations

  zero:Nat //constant, operation?
  one: Nat //constant, operation?
  …
  plus:Nat x Nat → Nat
  minus:Nat x Nat → Nat
  times:Nat x Nat → Nat

Primitive Domains

- What about natural number subtraction?
  What if we have subtract(three,five)?
- Using the algebra we can construct expressions that represent members of Nat e.g. plus(times(three, two), minus (one, zero)) which we determine (through simplification) represents the constant named seven.
- What about natural number division?
  What if we have division by zero?
  We need to add an extra element to Nat called error. What impact has this on the other operations? E.g. Plus(three, error).

Primitive Domains

- Truth values

  Domain Tr = B

  Operations
  - true: Tr
  - false:Tr
  - not: Tr → Tr
  - or: Tr x Tr → Tr
  - (_→ _ [] _): Tr xD xD → D (for some D)

- Simplify
  1. ((not (false)) or false
  2. (true or false) →(seven div three) [] zero
  3. not(not true) → false [] false or true

Primitive Domains

- Truth values

  Domain Tr = B

  Operations
  - true: Tr
  - false:Tr
  - not: Tr → Tr
  - or: Tr x Tr → Tr
  - (_→ _ [] _): Tr xD xD → D (for some D)

- Simplify
  1. ((not (false)) or false) TRUE
  2. (true or false) →(seven div three) [] zero TWO
  3. not(not true) → false [] false or true FALSE

Primitive Domains

- Additional Nat operations:
  - equals: Nat x Nat → Tr
  - lessthan: Nat x Nat → Tr
  - greaterthan: Nat x Nat → Tr
  - – Define using lambda notation
  - – Include error states
- Expressions:

  not(four equals (one plus three)) →
  (one greaterthan zero) [] ((five times two) lessthan zero)

Primitive Domains

Character strings

Domain String = the character strings from elements of **C**
(character domain including ``error'')
Operations
A, B, C, ... , Z: String
empty: String
error: String
concat: String x String → String
length: String → Nat
substr: String x Nat x Nat → String

Primitive Domains

- What happens if we try to evaluate
  substr ((A concat (B concat C)), one , four)
  We need to use error => add an error element to String and extend all other operations appropriately.

- The one element domain
  Domain Unit,
  Operations
  ( ) : Unit
  This domain may be used whenever an operation needs a dummy arguments e.g.let f: Unit → Nat be f(x) = one; thus f() = one.

Primitive Domains

- Computer store locations
  Domain Location, the address space in a computer store
  Operations
  first-locn: Location
  next-locn: Location → Location
  eq-locn: Location x Location →Tr
  less-l: Location x Location → Tr

  In adequate for defining the semantics of an assembly language, as an assembly lang. Allows random access of the locations in a store and treats locations as numbers. It works well for prog. Langs whose storage is allocated in static or stack like fashion.

Compound Domains

- Domain theory possesses a number of domain building constructions for creating new domains from existing ones.
- Each domain builder carries with it a set of operation builders for assembling and disassembling elements of the compound domain.
- Compound Domains include:
  - Product domains A x B
  - Sum domains A+ B
  - Function domains A → B
  - Lifted domains $A_{\perp} = A \cup \{\perp\}$

Product Domains

- The product domain takes two or more component domains and builds a domain of tuples from the components.
- The product domain builder x builds the domain A x B, a collection whose members are ordered pairs of the form (a,b), for a ∈ A and b ∈ B.
- Disassembly operators: fst: A x B → A, snd: A x B → B
- Assembly operator: if a ∈ A, and b ∈ B, then (a,b) is an element of A x B.
- An example of a semantic algebra built with the product construction follows:

Product Domains

Payroll information (name, payrate, hours)

Domain Payroll = String x Rat x Rat

Operations

newemp: String → Payroll

newemp(name) = (name, minwage, 0)

where minwage ∈ Rat and 0 = makerat(0)(1)

upd-payrate: Rat x Payroll → Payroll

upd-payrate(pay, emp) = (emp↓1, pay, emp ↓3)

upd-hours: Rat x Payroll → Payroll

upd-hours(hours, emp) = (emp↓1,emp↓2, addrat(hours)(emp↓3))

compute-pay: Payroll → Rat

compute-pay(emp) = multrat(emp ↓ 2)(emp ↓ 3)

Note: $(a_1, a_2, .., a_n) \downarrow i = a_i$

Example:

compute-pay(upd-hours(35, newemp ("j.Doe")))

= …

< expand newemp, then upd-hours, then simplify, then expand compute-pay> ….

…...

= multrat (minwage)(35)

Disjoint Union

- Disjoint Union Builder: + , builds A + B, a collection whose members are the elements of A and the elements of B, labelled to mark their origins, e.g. (zero, a) for an a ∈ A and (one, b) for a b ∈ B
- Assembly:
  - inA: A → A + B i.e. inA(a) = (zero,a)
  - inB: B → A + B i.e. inB(b) = (one,b)
- Disassembly: Cases operation which checks the tag of its argument, removes it and gives the argument to the proper operation.
  - If d is a value from A + B and f(x) = e1 and g(y) = e2 are the definitions of f:A → C and g: B → C, then cases d of isA(x) →e1 [] isB(x) → e2 end

Sum Domains

- Revised payroll information

Domain Payroll = String x (Day + Night) x Rat

where Day = Night = Rat

Operation: newemp: String → Payroll

newemp(n) = (n, inDay(minwage), 0)

// new emps are started on the day shift

move-to-day: Payroll → Payroll

move-to-day(emp) =

(emp ↓ 1,

cases emp ↓ 2 of

isDay(wage) → inDay(wage)

isNight(wage) → inDay(wage)end,

emp ↓ 3)

```
compute-pay: Payroll → Rat
  compute-pay(emp) =
        cases emp ↓ 2 of
        isDay(wage) → multrat(wage)(emp ↓ 3)
        isNight(wage) → multrat(1.5)
                (multrat(wage)(emp ↓ 3))
        end
```

Define move-to-night: Payroll → Payroll.

Example:

If jdoe is the expression
newemp ("j.Doe" = ("J,Doe", inDay(minwage), 0) and
jdoe-thirty is upd-hours(30,jdoe), then

compute-pay(upd-hours(30,jdoe)) = ……..
….
….
…
= minwage multrat 30

Sum Domains: Truth values as disjoint union

Domain Tr = TT + FF
where TT = Unit and FF = Unit
Operations:

```
true = inTT()
false = inFF()
not(t) = cases t of
        isTT() → inFF()
        [] isFF() → inTT()
end
or(t, u) = cases t of isTT() → inTT()
                []isFF() → (cases u of isTT() → inTT()
                                [] isFF() → inFF() end)
        end
```


Function Space

- Function Space Builder: →, collects the functions from a domain A to a domain B, creating the domain A →B.
- Disassembly: _ (_):(A → B) x A → B which takes an f ∈ A → Band an a ∈ A and produces f(a) ∈ B
- Extensionality: for any f and g in A → B, if for all a ∈ A , f(a) = g(a) then f = g
- Assembly: if e is an expression containing occurences of an identifier x, such that whenever a value a ∈ A replaces the occurences of x in e then the value [a/x] e ∈ B results, then (λx.e)is an element in A → B
- Note:
- [n → v] r abbreviates (λ m.m equal n → v []r(m))
  i.e. if this function **r** is applied to **n** the result is **v** otherwise is the result is **r** applied to **m.**

Lambda Calculus

- (λm.(λn.n times n)(m plus two))(one)
- (λm. λn.(m plus m) times n)(one)(three)
- (λm.λn.n + n)(m)
- (λp.λq.p + q) r+1

- Abstraction
- Bound Variables
- Free Variables
- Renaming Variables

Function Domains -Dynamic arrays

Domain Array = Nat A where A is a domain with an error element
Operations

newarray: Array	An empty array
newarray = λ n.error	Maps all of index elements to error
access: Nat x Array → A	Nat used to index array
access(n, r) = r(n)	Indexes its array argument r at position n
update: Nat **x** A **x** Array → Array	
update(n, v, r) = [n → v]r	Creates a new array that behaves just like r when indexed an any position but n. When indexed at position n, the new array produces the value v.

Dynamic arrays with curried operations

Domain Array = Nat $\rightarrow$ A
(where A is a domain with an error element)

Operations
newarray: Array
newarray = λ n.error

access: Nat $\rightarrow$ Array $\rightarrow$ A
access = λ n. λ r.r(n)

update: Nat $\rightarrow$ A $\rightarrow$ Array $\rightarrow$ Array
update = λ n. λ v. λ r.[n $\rightarrow$ v]r

Lifted Domains

- The lifting domain builder $(\ )_\perp$ creates the domain $A_\perp$ a collection of the members of A plus an additional distinguished element $\perp$
- The disassembly operation builder converts an operation on A to one on $A_\perp$ for $(\lambda x.e):A \rightarrow B_\perp$:
  $(\lambda x.e):A_\perp \rightarrow B_\perp$:is defined as $(\lambda x.e)\perp = \perp$
  $(\lambda x.e)a = [a/x]e$ for $a \neq \perp$
- An operation that maps a $\perp$ argument to a $\perp$ is called strict. Operations that map $\perp$ to a proper element are called nonstrict.

Examples:

- Strict
  $(\lambda m.zero)((\lambda\ n.one)\perp)$
  $= (\lambda m.zero)\perp$ (by strictness)
  $= \perp$
  In this example we simplify the argument to determine if it is proper or improper. This is a call by value evaluation.
- Non Strict
  $(\lambda p.zero)((\lambda\ n.one)\perp)$
  $= [(\lambda\ n.one)\perp/p]zero$ (by definition of application)
  = zero
  Here there is no need to simplify the argument $((\lambda\ n.one)\perp)$ first

Notation:

- We use the following abbreviation:
  (let x = e1 in e2) for $(\lambda x.e2)e1$
- Call this a **let** expression. It makes strict applications more readable because its "argument first" appearance matches the argument first simplification strategy that must be used.

Lifted Domains: Unsafe arrays of unsafe values

Domain Unsafe = $\text{Array}_\perp$ where Array = Nat $\rightarrow$ Tr' and Tr' = $(B \cup \{error\})_\perp$
Operations
new-unsafe: Unsafe
new-unsafe = newarray
access-unsafe: $\text{Nat}_\perp \rightarrow$ Unsafe $\rightarrow$ Tr'
access-unsafe = $\underline{\lambda}$ n. $\underline{\lambda}$ r.(access n r)
update-unsafe $\text{Nat}_\perp \rightarrow$ Tr '$\rightarrow$ Unsafe $\rightarrow$ Unsafe
update-unsafe = $\underline{\lambda}$ n. $\underline{\lambda}$ t. $\underline{\lambda}$ r.(update n t r)

- Note: Indices and elements may be improper! Change this!

Recursive Function Definitions

Recursive function definitions need not define a function uniquely! The specification q(x) = (x equals zero) $\rightarrow$ one [] q(x plus one) defines a function in $N \rightarrow N_\perp$. The following functions all satisfy that specification:

- f 1 (x) = one if x = zero
  $\perp$ otherwise
- f 2 (x) = one if x = zero
  two otherwise
- f 3 (x) = one

- Verify that f3 is a meaning of q:
  for any n ∈ Nat, n equals zero → one [] f3(n plus one)
  = n equals zero → one [] one
  = one
  = f3(n)

  Similarly we can show that f1 and f2 are meanings of q.
  So which of these functions does q really stand for, if any?
  Unfortunately, the tools as currently developed are not sophisticated enough to answer this question (see section 6).

Recursive Domain Definitions

- Certain program language features require domains whose structure is defined in terms of themselves
  - e.g. Alist = Unit + (A x Alist) defines a domain of linear lists of A elements.
- Like recursively defined operations, a domain may not be uniquely defined by a recursive definition.

Section 4: Basic Structure of Denotational Definitions

Here we present the format for Denotational definitions using the abstract syntax and semantic algebra formats to define the appearance and the meaning of the language. The two are connected by a function called the **valuation** function.

Valuation Function

- The valuation function maps a language's abstract syntax structures to meanings drawn from semantic domains.
- The domain of a valuation function is the set of derivation trees of a language.
- The valuation function is defined structurally
- It determines the meaning of a derivation tree by determining the meanings of its subtrees and combining them into a meaning for the entire tree.

Format of a denotational definition

- A denotational definition of a language consists of three parts:
  - abstract syntax definition
  - semantic algebras
  - valuation function - a collection of functions, one for each syntactic domain.
  - E.g. Figure 4.1
  - What is the meaning of 101 i.e.**B [**|101 **|] ?**

A Calculator Language

- Buttons and display screen,
- Single memory cell,
- Conditional evaluation feature.
- Input Display
  - Press ON
  - Press (4 + 1 2) * 2
  - Press TOTAL (prints 32)
  - Press 1 + LASTANSWER
  - Press TOTAL (prints 33)
  - Press IF LASTANSWER + 1 , 0 , 2 + 4
  - Press TOTAL (prints 6)
  - Press OFF
- (See Schmidt, Figure 4.3)

Basic Structure of Denotational Definitions Observations

- Global data structures are modelled as arguments to valuation functions. No 'global variables' for functions e.g. memory cell of **S**
- Meaning of a syntactic construct can be a function e.g. S's functionality states that the meaning of an expression sequence is a function from a memory cell to a list of numbers.

S Rule: S[|E TOTAL S|]

Calculator actions:

- Evaluate **[| E |]** using cell n producing value n'
- Print n' on the display.
- Place n' into the memory cell.
- Evaluate the rest of sequence **[| S |]** using the cell.

- How are these represented in the semantic domain?

- Note: right-hand side of equation is a mathematical value!

Simplification

- **P**[|ON 2+1 TOTAL IF LA,2,0 TOTAL OFF|]

= **S** [| 2+1 TOTAL IF LA, 2, 0 TOTAL OFF |] (zero)
= let n' = **E** [| 2+1 |](zero)
in n' cons S [| IF LA , 2 , 0 TOTAL OFF |](n')

= let n' = three in n' cons S [| IF LA , 2 , 0 TOTAL OFF |](n')
= three cons S [| IF LA , 2 , 0 TOTAL OFF |](three)

= three cons (E [| IF LA , 2 , 0 |](three) cons nil)
= three cons (zero cons nil)

- E [| IF LA , 2 , 0 |](three)
= E [| LA |](three) equals zero → E [| 2 |](three) [] E [| 0 |](three)
= three equals zero → two [] zero
= false → two [] zero = zero

Simplification

- Each simplification step preserves meaning.
- Goal is to produce equivalent expression whose meaning is more obvious than the meaning of the original.
- Simplification process shows how program operates
- Denotational definition is used as a specification for the calculator.
- Denotational definition plus simplification strategy shows a possible implementation of the calculator.