# Understanding Computation - From Simple Machines to Impossible Programs

## The Meaning of "Meaning"

In linguistics, semantics is the study of the connection between words and their meanings: the word "dog" is an arrangement of shapes on a page, or a sequence of vibrations in the air caused by someone's vocal cords, which are very different things from an actual dog or the idea of dogs in general.

## How to specify a language

1. asyntax

2. a semantics

In the context of programming language theory, the word semantics is usually treated as singular: we describe the meaning of a language by giving it a semantics

## Three Ways to Specify a language

### An Interpreter

Plenty of languages don't have an official written specification, just a working interpreter or compiler

#### MRI

If any piece of Ruby documentation disagrees with the actual behavior of MRI, it's the documentation that's wrong

PHP and Perl 5

### Prose Specification

Another way of describing a programming language is to write an official prose specification, usually in English.

C++, Java, and ECMAScript

### Mathematical Techniques

A third alternative is to use the mathematical techniques of formal semantics to precisely describe the meaning of a programming language.

## Syntax

a collection of rules that describe what kind of character strings may be considered valid programs in that language

### Parser

a program that can read a character string representing a program, check it against the syntax rules to make sure it's valid, and turn it into a structured representation of that program suitable for further processing.

### Abstract Syntax Tree

abstract syntax tree (AST)

a representation of the source code that discards incidental detail like whitespace and focuses on the hierarchical structure of the program.

## Chapter 2. The Meaning of Programs

Computer programming isn't really about programs, it's about ideas.

## Expressions

The purpose of an expression is to be evaluated to produce another expression

### Number, Add, and Multiply

```
class Number < Struct.new(:value)
end

class Add < Struct.new(:left, :right)
end

class Multiply < Struct.new(:left, :right)
end
```

```
>> Add.new(
     Multiply.new(Number.new(1), Number.new(2)),
     Multiply.new(Number.new(3), Number.new(4))
   )
=> #<struct Add left=#<struct Multiply left=#<struct Number value=1>, right=#<struct Number value=2>>, right=#<struct Multiply left=#<struct Number value=3>, right=#<struct Number value=4>>>
```

### reducible?

```
class Number
  def reducible?
    false
  end
end

class Add
  def reducible?
    true
  end
end

class Multiply
  def reducible?
    true
  end
end


>> Number.new(1).reducible?
=> false
>> Add.new(Number.new(1), Number.new(2)).reducible?
=> true
```

### reduce

- If the addition's left argument can be reduced, reduce the left argument.
- If the addition's left argument can't be reduced but its right argument can, reduce the right argument.
- If neither argument can be reduced, they should both be numbers, so add them together.

```
class Add
  def reduce
    if left.reducible?
      Add.new(left.reduce, right)
    elsif right.reducible?
      Add.new(left, right.reduce)
    else
      Number.new(left.value + right.value)
    end
  end
end

class Multiply
  def reduce
    if left.reducible?
      Multiply.new(left.reduce, right)
    elsif right.reducible?
      Multiply.new(left, right.reduce)
    else
      Number.new(left.value * right.value)
    end
  end
end
```

```
>> expression = Add.new(
     Multiply.new(Number.new(1), Number.new(2)),
     Multiply.new(Number.new(3), Number.new(4))
   )
=> #<struct Add left=#<struct Multiply left=#<struct Number value=1>, right=#<struct Number value=2>>, right=#<struct Multiply left=#<struct Number value=3>, right=#<struct Number value=4>>>
>> expression.reducible?
=> true
>> expression = expression.reduce
=> #<struct Add left=#<struct Number value=2>, right=#<struct Multiply left=#<struct Number value=3>, right=#<struct Number value=4>>>
>> expression.reducible?
=> true
>> expression = expression.reduce
=> #<struct Add left=#<struct Number value=2>, right=#<struct Number value=12>>
>> expression.reducible?
=> true
>> expression = expression.reduce
=> #<struct Number value=14>
>> expression.reducible?
=> false
```

### maintaining a piece of state

```
class Machine < Struct.new(:expression)
  def step
    self.expression = expression.reduce
  end

  def run
    while expression.reducible?
      puts expression
      step
    end


    puts expression
  end
end
```

```
>> Machine.new(
     Add.new(
       Multiply.new(Number.new(1), Number.new(2)),
       Multiply.new(Number.new(3), Number.new(4))
     )
   ).run
1 * 2 + 3 * 4
2 + 3 * 4
2 + 12
14
=> nil
```

### Boolean true and false; Boolean and, or, and not

```
class Boolean < Struct.new(:value)
  def to_s
    value.to_s
  end

  def inspect
    "«#{self}»"
  end

  def reducible?
    false
  end
end
```

```
class LessThan < Struct.new(:left, :right)
  def to_s
    "#{left} < #{right}"
  end

  def inspect
    "«#{self}»"
  end

  def reducible?
    true
  end

  def reduce
    if left.reducible?
      LessThan.new(left.reduce, right)
    elsif right.reducible?

      LessThan.new(left, right.reduce)
    else
      Boolean.new(left.value < right.value)
    end
  end
end
```

```
>> Machine.new(
     LessThan.new(Number.new(2), Add.new(Number.new(2), Number.new(2)))
   ).run
2 < 2 + 2
2 < 4
=> nil
```

### variables

## Operational Semantics

defining rules for how its programs execute on some kind of device

This device is often an abstract machine

## Small-Step

Small-Step Semantics

### DoNothing

```
class DoNothing ❶
  def to_s
    'do-nothing'
  end

  def inspect
    "«#{self}»"
  end

  def ==(other_statement) ❷
    other_statement.instance_of?(DoNothing)
  end

  def reducible?
    false
  end
end
```

### ==

We want to be able to compare any two statements to see if they're equal. The other syntax classes inherit an implementation of #== from Struct, but DoNothing has to define its own.

### Assign

- If the assignment's expression can be reduced, then reduce it, resulting in a reduced assignment statement and an unchanged environment.
- If the assignment's expression can't be reduced, then update the environment to associate that expression with the assignment's variable, resulting in a «do-noth ing» statement and a new environment.

```
class Assign < Struct.new(:name, :expression)
  def to_s
    "#{name} = #{expression}"
  end

  def inspect
    "«#{self}»"
  end

  def reducible?
    true
  end

  def reduce(environment)
    if expression.reducible?
      [Assign.new(name, expression.reduce(environment)), environment]
    else
      [DoNothing.new, environment.merge({ name => expression })]
    end
  end
end
```

#### New Machine

```
class Machine < Struct.new(:statement, :environment)
  def step
    self.statement, self.environment = statement.reduce(environment)
  end

  def run
    while statement.reducible?
      puts "#{statement}, #{environment}"
      step
    end

    puts "#{statement}, #{environment}"
  end
end
```

#### Output

```
>> Machine.new(
     Assign.new(:x, Add.new(Variable.new(:x), Number.new(1))),
     { x: Number.new(2) }
   ).run
x = x + 1, {:x=>2}
x = 2 + 1, {:x=>2}
x = 3, {:x=>2}
do-nothing, {:x=>3}
=> nil
```

### If

if (x) { y = 1 } else { y = 2 }

- If the condition can be reduced, then reduce it, resulting in a reduced conditional statement and an unchanged environment.
- If the condition is the expression «true», reduce to the consequence statement and an unchanged environment.
- If the condition is the expression «false», reduce to the alternative statement and an unchanged environment.

```
class If < Struct.new(:condition, :consequence, :alternative)
  def to_s
    "if (#{condition}) { #{consequence} } else { #{alternative} }"
  end

  def inspect
    "«#{self}»"
  end

  def reducible?
    true
  end

  def reduce(environment)
    if condition.reducible?
      [If.new(condition.reduce(environment), consequence, alternative), environment]
    else
      case condition
      when Boolean.new(true)
        [consequence, environment]
      when Boolean.new(false)
        [alternative, environment]
      end
    end
  end
end
```

#### If

```
class If < Struct.new(:condition, :consequence, :alternative)
  def to_s
    "if (#{condition}) { #{consequence} } else { #{alternative} }"
  end

  def inspect
    "«#{self}»"
  end

  def reducible?
    true
  end

  def reduce(environment)
    if condition.reducible?
      [If.new(condition.reduce(environment), consequence, alternative), environment]
    else
      case condition
      when Boolean.new(true)
        [consequence, environment]
      when Boolean.new(false)
        [alternative, environment]
      end
    end
  end
end
```

#### Output

```
>> Machine.new(
     If.new(
       Variable.new(:x),
       Assign.new(:y, Number.new(1)),
       Assign.new(:y, Number.new(2))
     ),
     { x: Boolean.new(true) }
   ).run
if (x) { y = 1 } else { y = 2 }, {:x=>true}
if (true) { y = 1 } else { y = 2 }, {:x=>true}
y = 1, {:x=>true}
do-nothing, {:x=>true, :y=>1}
=> nil
```

### sequence

- If the first statement is a «do-nothing» statement, reduce to the second statement and the original environment.
- If the first statement is not «do-nothing», then reduce it, resulting in a new sequence (the reduced first statement followed by the second statement) and a reduced environment.

```
class Sequence < Struct.new(:first, :second)
  def to_s
    "#{first}; #{second}"
  end

  def inspect
    "«#{self}»"
  end

  def reducible?
    true
  end

  def reduce(environment)
    case [first]

    when [DoNothing.new
      [second, environment]
    else
      [reduced_first, reduced_environment] = first.reduce(environment)
      [Sequence.new(reduced_first, second), reduced_environment]
    end
  end
end
```

#### Sequence

```
class Sequence < Struct.new(:first, :second)
  def to_s
    "#{first}; #{second}"
  end

  def inspect
    "«#{self}»"
  end

  def reducible?
    true
  end

  def reduce(environment)
    case [first]

    when [DoNothing.new
      [second, environment]
    else
      [reduced_first, reduced_environment] = first.reduce(environment)
      [Sequence.new(reduced_first, second), reduced_environment]
    end
  end
end
```

#### Output

```
>> Machine.new(
     Sequence.new(
       Assign.new(:x, Add.new(Number.new(2), Number.new(2))),
       Assign.new(:y, Add.new(Variable.new(:x), Number.new(3)))
     ),
     {}
   ).run
x = 2 + 2; y = x + 3, {}
x = 4; y = x + 3, {}
do-nothing; y = x + 3, {:x=>4}
y = x + 3, {:x=>4}
y = 4 + 3, {:x=>4}
do-nothing, {:x=>4, :y=>7}
=> nil
```

### while

- Reduce «while (condition) { body }» to «if (condition) { body; while (condi tion) { body } } else { do-nothing }» and an unchanged environment.

```
class While < Struct.new(:condition, :body)
  def to_s
    "while (#{condition}) { #{body} }"
  end

  def inspect
    "«#{self}»"
  end

  def reducible?
    true
  end

  def reduce(environment)
    [If.new(condition, Sequence.new(body, self), DoNothing.new), environment]
  end
end
```

#### While

```
class While < Struct.new(:condition, :body)
  def to_s
    "while (#{condition}) { #{body} }"
  end

  def inspect
    "«#{self}»"
  end

  def reducible?
    true
  end

  def reduce(environment)
    [If.new(condition, Sequence.new(body, self), DoNothing.new), environment]
  end
end
```

#### Output

```
>> Machine.new(
     While.new(
       LessThan.new(Variable.new(:x), Number.new(5)),
       Assign.new(:x, Add.new(Variable.new(:x), Number.new(1)))
     ),
     { x: Number.new(0) }
   ).run
while (x < 5) { x = x + 1 }, {:x=>0}
if (x < 5) { x = x + 1; while (x < 5) { x = x + 1 } } else { do-nothing }, {:x=>0}
if (0 < 5) { x = x + 1; while (x < 5) { x = x + 1 } } else { do-nothing }, {:x=>0}
x = x + 1; while (x < 5) { x = x + 1 }, {:x=>0}
x = 0 + 1; while (x < 5) { x = x + 1 }, {:x=>0}
x = 1; while (x < 5) { x = x + 1 }, {:x=>1}
while (x < 5) { x = x + 1 }, {:x=>1}
if (x < 5) { x = x + 1; while (x < 5) { x = x + 1 } } else { do-nothing }, {:x=>1}
```

#### Pass self to Sequence

[If.new(condition, Sequence.new(body, ***self***), DoNothing.new), environment]

## Correctness

## Applications

## Big-Step Semantics