

# Semantics of programming languages

## Informatics 2A: Lecture 27

John Longley

School of Informatics  
University of Edinburgh  
als@inf.ed.ac.uk

24 November 2015

## Two parallel pipelines

A large proportion of the course thus far can be organised into two parallel language processing pipelines.

### Formal Language

Program text  
↓  
Lexemes  
↓  
Syntax tree  
↓  
(Annotated)  
syntax tree

lexing  
parsing  
type checking

### Natural Language

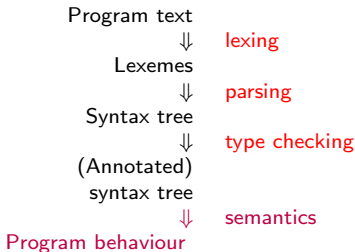
Text  
↓  
Tagged words  
↓  
Parse tree  
↓  
(Annotated)  
parse tree  
↓  
Logical form

segmentation, tagging  
parsing  
agreement checking  
semantics

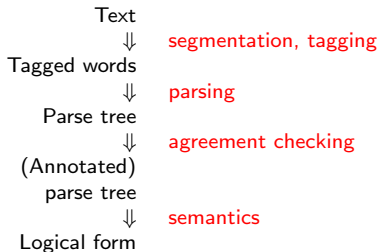
## Two parallel pipelines

A large proportion of the course thus far can be organised into two parallel language processing pipelines.

### Formal Language



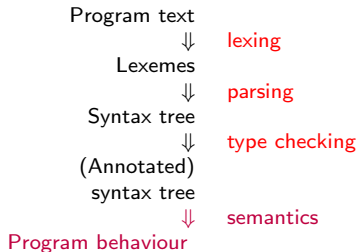
### Natural Language



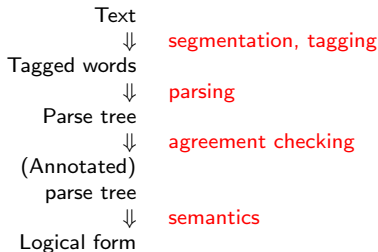
## Two parallel pipelines

A large proportion of the course thus far can be organised into two parallel language processing pipelines.

### Formal Language



### Natural Language



Today we look at methods of specifying program behaviour.

## Semantics for programming languages

The **syntax** of NLs (as described by CFGs etc.) is concerned with what sentences are **grammatical** and what structure they have, whilst their **semantics** are concerned with what sentences **mean**.

A similar distinction can be made for programming languages. Rules associated with lexing, parsing and typechecking concern the form and structure of legal programs, but say nothing about what programs should **do** when you run them.

The latter is what **programming language semantics** is about. It thus concerns the later stages of the **formal language processing pipeline**.

## Specification vs. implementation

In principle, one way to give a semantics (or ‘meaning’) for a programming language is to provide a **working implementation** of it, e.g. an interpreter or compiler for the language.

However, such an implementation will probably consist of thousands of lines of code, and so isn’t very suitable as a readable definition or **reference specification** of the language.

The latter is what we’re interested in here. In other words, we want to fill the blank in the following table:

	Specification	Implementation
Lexical structure	Regular exprs.	Lexer impl.
Grammatical structure	CFGs	Parser impl.
Execution behaviour	???	Interpreter/compiler

## Semantic paradigms

We'll look at two styles of formal programming language semantics:

- **Operational semantics.** Typically consists of a bunch of rules for 'executing' programs given by syntax trees. Oriented towards **implementations** of the language. indeed, an op. sem. often gives rise immediately to a 'toy implementation'.
- **Denotational semantics.** Typically consists of a **compositional** description of the meaning of program phrases (close in spirit to what we've seen for NLs). Oriented towards **mathematical reasoning** about the language and about programs written in it. May be 'executable' or not.

These two styles are complementary: ideally, it's nice to have both. There are also other styles (e.g. **axiomatic semantics**), but we won't discuss them here.

## Micro-Haskell: recap

We use Micro-Haskell (recall Lecture 13 and Assignment 1) as a vehicle for introducing the methods of operational and denotational semantics.

The format of MH declarations is illustrated by:

```
div :: Integer -> Integer -> Integer ;  
div x y = if x < y then 0 else 1 + div (x + -y) y ;
```

This declares a function `div`, of the type specified, such that, when applied to two (non-negative) integer literals  $\overline{m}$  and  $\overline{n}$ , the function application

$$\text{div } \overline{m} \ \overline{n}$$

returns, as result, the integer literal representing the integer division of  $m$  by  $n$ .



## Semantic paradigms in the case of MH

### Operational semantics:

This explains the **computational process** by which MH calculates the value of a function application, such as  $\text{div } \overline{m} \ \overline{n}$ .

### Denotational semantics:

This defines a mathematical **denotation**

$$\llbracket \text{div} \rrbracket \in \llbracket \text{Integer} \rightarrow \text{Integer} \rightarrow \text{Integer} \rrbracket$$

**Roughly**,  $\llbracket \text{Integer} \rightarrow \text{Integer} \rightarrow \text{Integer} \rrbracket$  is the set of binary functions on integers, and  $\llbracket \text{div} \rrbracket$  is the integer-division function.

**In reality**, denotational semantics is more complicated than this.

# Operational semantics

We model the execution behaviour of programs as a series of **reduction steps**.

E.g. for Micro-Haskell:

```
      if 4+5 <= 8 then 4 else 6+7
→→  if 9 <= 8 then 4 else 6+7
→→  if False then 4 else 6+7
→→  6+7
→→  13
```

A (small-step) operational semantics is basically a bunch of rules for performing such reductions.

## More complex example

Consider the Micro-Haskell declaration

$$f \ x \ y = x+y+x \ ;$$

This effectively introduces the definition

$$f = \lambda x. \lambda y. \ x+y+x$$

Now consider the evaluation of  $f \ 3 \ 4$ :

$$\begin{aligned} f \ 3 \ 4 &\rightarrow (\lambda x. \lambda y. \ x+y+x) \ 3 \ 4 \\ &\rightarrow (\lambda y. \ 3+y+3) \ 4 \\ &\rightarrow 3+4+3 \\ &\rightarrow 7 + 3 \\ &\rightarrow 10 \end{aligned}$$

Notice that two of these steps are  $\beta$ -reductions!

## Operational semantics for Micro-Haskell: general rules

Suppose  $E$  is a **runtime environment** associating a definition to each function symbol, e.g.  $E(f) = \lambda x. \lambda y. x + y + x$ .

Also let  $v$  range over **variables** of MH, and write  $\bar{n}$  to mean the integer literal for  $n$ .

Relative to  $E$ , we can define  $\rightarrow$  as follows:

- $v \rightarrow E(v)$  ( $v$  a variable defined in  $E$ )
- $(\lambda v. M)N \rightarrow M[v \mapsto N]$  ( $\beta$ -reduction)
- $\bar{m} + \bar{n} \rightarrow \overline{m + n}$ , and similarly for other infixes.
- if True then  $M$  else  $N \rightarrow M$
- if False then  $M$  else  $N \rightarrow N$

Continued on next slide ...

## Operational semantics for Micro-Haskell (continued)

Let's say a term  $M$  is a **value** if it's an integer literal, a boolean literal, or a  $\lambda$ -abstraction. Let  $V$  range over values,

**Intuition:** values are terms that can't be reduced any further. We try to reduce all other terms to values.

To complete the definition of  $\rightarrow$ , we decree that if  $M \rightarrow M'$  then:

- $MN \rightarrow M'N$
- $M \odot N \rightarrow M' \odot N$  ( $\odot$  any infix symbol)
- $V \odot M \rightarrow V \odot M'$  (ditto)
- if  $M$  then  $N$  else  $P \rightarrow$  if  $M'$  then  $N$  else  $P$

We then say  $M \rightarrow^* V$  (" $M$  evaluates to  $V$ ") if there's a sequence

$$M \equiv M_0 \rightarrow M_1 \rightarrow \cdots \rightarrow M_r \equiv V$$

That defines the intended behaviour of Micro-Haskell programs. It's also how the Assignment 1 **evaluator** for MH works.

## Two questions

Consider the following MH program.

```
e :: Integer ;  
e = e ;  
  
f :: Integer -> Integer ;  
f x = if True then 2+3 else 1+x ;
```

What happens if the expressions below are evaluated?

e                      f e

- 1 There is a parse error.
- 2 There is a type error.
- 3 The evaluator goes into a loop.
- 4 The evaluator terminates and produces an integer as result.
- 5 The evaluator terminates and produces a different result.

## Operational semantics: further remarks

What happens if we encounter an expression that isn't a value but can't be reduced? E.g. `5 True`, or `( $\lambda x.x$ )+4` ?

!!! If our original program typechecks, this can never happen !!!

Indeed, it can be proved that:

- if  $M$  can be typed, either it's a value or it can be reduced;
- if  $M$  has type  $t$  and  $M \rightarrow M'$ , then  $M'$  has type  $t$ .

That's one reason why type systems are so valuable: they can guarantee programs won't derail at runtime.

The general form of operational semantics we've described is immensely flexible. It works beautifully for functional languages like MH. But it can also be adapted to most other kinds of programming language.

## Denotational semantics

An operational semantics provides a kind of idealized implementation of the language in terms of symbolic rules.

That's fine, but doesn't give much 'structural' understanding. Conceptually and mathematically, it is more satisfying to assign **meaning** to (parts of) a program — in roughly the way that mathematical expressions (or indeed NL expressions) have meaning.

This is the idea behind denotational semantics: associate a **denotation**  $\llbracket P \rrbracket$  to each program phrase  $P$  in a compositional way.



## Denotational semantics for MH: first attempt

Let's try interpreting MH types by **sets** in a natural way:

$$\llbracket \text{Integer} \rrbracket = \mathbb{Z} \qquad \llbracket \text{Bool} \rrbracket = \mathbb{B} = \{T, F\}$$

$$\llbracket \sigma \rightarrow \tau \rrbracket = \llbracket \tau \rrbracket^{\llbracket \sigma \rrbracket} \quad (\text{set of all functions from } \llbracket \sigma \rrbracket \text{ to } \llbracket \tau \rrbracket)$$

A term  $M :: \tau$  will receive a denotation  $\llbracket M \rrbracket \in \llbracket \tau \rrbracket$ . More accurately, if  $M :: \tau$  is a term in the **type environment**

$\Gamma = \langle x_1 :: \sigma_1, \dots, x_n :: \sigma_n \rangle$ , its denotation will be a function

$$\llbracket M \rrbracket_{\Gamma} : \llbracket \sigma_1 \rrbracket \times \dots \times \llbracket \sigma_n \rrbracket \rightarrow \llbracket \tau \rrbracket$$

We define  $\llbracket M \rrbracket_{\Gamma}$  **compositionally**. E.g. writing  $\vec{a}$  for  $\langle a_1, \dots, a_n \rangle$ :

$$\llbracket \bar{n} \rrbracket_{\Gamma} : \vec{a} \mapsto n$$

$$\llbracket x_i \rrbracket_{\Gamma} : \vec{a} \mapsto a_i$$

$$\llbracket M + N \rrbracket_{\Gamma} : \vec{a} \mapsto \llbracket M \rrbracket_{\Gamma}(\vec{a}) + \llbracket N \rrbracket_{\Gamma}(\vec{a})$$

$$\llbracket M N \rrbracket_{\Gamma} : \vec{a} \mapsto \llbracket M \rrbracket_{\Gamma}(\vec{a})(\llbracket N \rrbracket_{\Gamma}(\vec{a})), \quad \text{etc.}$$

## Denotational semantics for MH: the challenge

That works well as far as it goes. The problem comes when we try to interpret **recursive** definitions, e.g.

```
div =  $\lambda x.\lambda y.$ if  $x < y$  then 0 else  $1 + \text{div } (x + -y) y$  ;
```

Here we'd end up trying to define  $\llbracket \text{div} \rrbracket$  in terms of itself!

Our simple 'set-theoretic' interpretation can't make sense of this. One needs an alternative interpretation that builds in some deeper mathematical properties in order to allow 'circular definitions'. E.g. one can interpret the whole of MH using

- **complete partial orders** (non-executable semantics)
- **game models** (executable semantics)

This is where denotational semantics gets interesting, and where, reluctantly, we move on from MH to something simpler ...

## Denotational semantics for regular expressions

Let's turn to an easier example. Recall our (meta)language of **regular expressions**:

$$R \rightarrow \epsilon \mid \emptyset \mid a \mid RR \mid R + R \mid R^*$$

In fact, we've already met **two** good den. sems. for this!

- $\llbracket R \rrbracket_1 = \mathcal{L}(R)$ , the language (i.e. set of strings) defined by  $R$ .
- $\llbracket R \rrbracket_2$  = the particular ( $\epsilon$ -)NFA for  $R$  constructed by the methods of Lecture 5.

Both of these are defined **compositionally**: e.g.  $\mathcal{L}(R + R')$  is defined as  $\mathcal{L}(R) \cup \mathcal{L}(R')$ , and the standard NFA for  $R + R'$  is constructed out of NFAs for  $R$  and  $R'$ . Note that:

- $\llbracket - \rrbracket_1$  is **more abstract** than  $\llbracket - \rrbracket_2$ : can have  $\llbracket R \rrbracket_2 \neq \llbracket R' \rrbracket_2$  but  $\llbracket R \rrbracket_1 = \llbracket R' \rrbracket_1$ . So  $\llbracket - \rrbracket_1$  is more useful for arguing that two regular expressions are 'equivalent'.
- However,  $\llbracket - \rrbracket_2$  is naturally **executable**, while  $\llbracket - \rrbracket_1$  is not.

## Summary

- Formal semantics can be used to give a concise and precise **reference specification** for the intended behaviour of programs.
- Operational semantics is nowadays widely used. Denotational semantics gets quite mathematical and is at present more of a research topic.
- Operational semantics, and some kinds of denotational semantics, also offer a starting-point for building working **implementations** of the language.
- Denotational semantics also offers a framework for **proving** things about programs. E.g. if  $\llbracket P \rrbracket = \llbracket P' \rrbracket$ , that shows that  $P$  can be replaced by  $P'$  *in any program context* without changing the program's behaviour.
- Ideas from both op. and den. semantics have had a significant effect on the **design** of programming languages.