History of Programming Languages

19 Jan 2021

- What is HOPL about
- The Dawn of PL
- Sunrise
- A 30min crash course on PPL
- How this course is run

(1) History of the programming language research area
- ideas, concepts, techniques

vs

(2) History of individual programming languages and how they relate
- impact on developers
- and the industry as a whole

This seminar is about (1).

It assumes a course on PPL.

Dawn, part 1

1920s, 1930s

- Gödel's implicit PL
- Church's λ-calculus
- Turing's machines

+ fundamental concepts of computing and <u>programming</u>

+ proof <u>techniques</u> for PL

- all languages are implicit (but surprisingly modern)

Dawn, part 2

1940s, 1950s

- Zuse's machines
- van Neumann machine
- Eniac
- a multitude of hardware and simple "assembly" PLs

+ fundamental concepts of implementation / compilation

- arguably a plan-less, bottom-up effort that in many ways headed in wrong direction

Sunrise

~ 1958 – 1960

- FORTRAN
- ALGOL 60
- COBOL
- LISP
- SNOBOL

+ "high" level PLs with a need for implementation, analysis, teaching, and standardization

+ PL (as in area) emerges from ALGOL and LISP; History begins

The Origin Story

Backus and Naur

McCarthy

Landin

Morris (MIT)

\+ think about PL as a whole and what it takes to design, build, implement, and standardize

demands thinking about PL w/o reference to an implementation

Footnotes

Dijkstra
Hoare
Floyd

think about individual programs

the act of programming and reasoning about a program at a time

in the process, they are forced to consider a PL but only fragments

One More Footnote

Knuth

program = algorithm

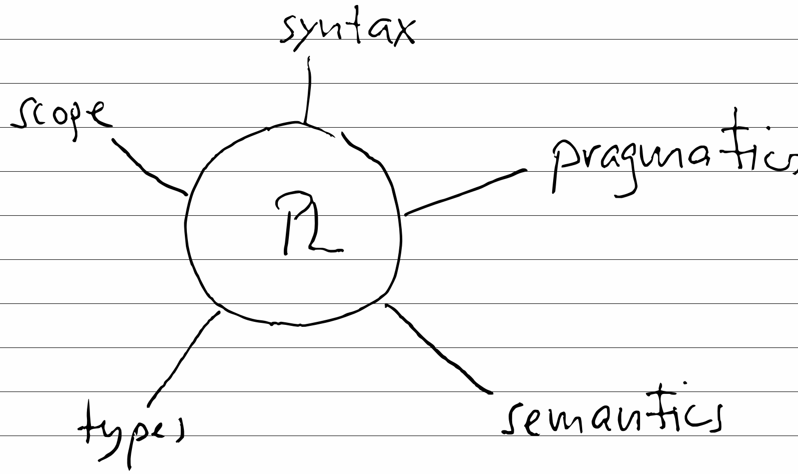
attribute grammars

ARGH!

re-appears time & again

see syntax-parse

Fundamental Concepts



SYNTAX : BNF

$\Downarrow$

generate scanners & parsers

SCOPE : NONE, DYNAMIC, STATIC } stack vs registers, ease of push/pop

TYPES: (1) data representation
hints for compilers
(I, J vs X, Y)

vs

(2) Program consistency
for programmers
(STLC)

SEMANTICS: BNF specifies syntax
and we derive PARSERS

vs

How to specify
semantics and
derive code generators

≅!

???

Pragmatics: How to use a PL properly?

→ Morris '68, next to
semantics

More on SEMANTICS

as early as 1968,
Morris explained that the
semantic essence of a PL
is observational equivalence

Def $e \cong_{A} e'$ iff for all
program contexts C,
$eval(C[e]) = eval(C[e'])$

if both are defined, mathematically equal

$\Longleftrightarrow$ regardless of how eval
(the semantics) is specified

Thm $\cong$ is canonical.

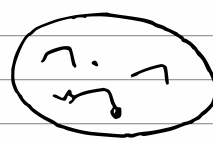
Critical for reasoning & compilation.

(see lecture 2)

More on PRAGMATICS

Morris also points out that in terms of usage, a PL's pragmatics and its explanation to programmers is the most critical aspect.

To this day, PL has nothing to say wrt. pragmatics.

 !

Very, very little (see SE)

The FUNDAMENTAL ARTIFACT

PL settled surprisingly quickly on
the LAMBDA CALCULUS

as the "thing" to study:

- a model of the essence of PL
- a meta-language for semantics
- an actual programming language
- a playground for experimenting with types

not immediately, but soon

- meta-theory techniques

Emerging Semantic Specs.

interpreters

recursive functions
from syntax to values
McCarthy '61

abstract machines

state machines w/
transitions and 'eval'
as "acceptance function"
Landin '63/64

uses: specify ALGOL and LISP

criticism: ad hoc, details,
meaning of the
meta-medium

Reynolds, '72

definitional interpreters
are bad because they
leave a lot of aspects
implicit (scope, parameters)
and thus inherit implicitly
from the meta-language

$\rightsquigarrow$ CPS

indeed, it assumes
an understanding
of this PL

$\hookrightarrow$ push problem
back by 1 level

So what does λ-calculus mean
when we say object of study

Syntax

$\boxed{\text{ISWIM}}$

Landin

typen?

e	$=$	$\lambda x . e$	abs
	$\mid$	x	var
	$\mid$	$e\ e$	app
	$\mid$	n	data
	$\mid$	op $e \ldots$	op. on data

\+ control
J : Landin
escape : Reynolds

\+ imperative assignment
(a) ref , :=, !
(b) variable assignment

Types

$\boxed{\text{PAL}}$

Morris

τ	$=$	int	(for data)
	$\mid$	$\tau \to \tau$	(functions)

plus <u>fix</u> for explicit recursion

Interpreter

$$eval : \Lambda \longrightarrow \mathbb{Z} + \text{"closure"}$$

$$eval(e) = \begin{cases} n & \text{if } e = n \\ \vdots & \\ eval(e_1) \doteq eval(e_2) & \text{if } e = e_1 e_2 \end{cases}$$

What does $\doteq$ mean?

abstract machine

$$eval : \Lambda \longrightarrow \mathbb{Z} + \text{"closure"}$$

$$eval = load \mid run \mid unload$$

$$run : \langle \underline{S}, \underline{E}, \underline{C}, \underline{D} \rangle \longrightarrow \langle S', E', C', D' \rangle$$

until final state

Why 4 "registers"

How to specify/understand states

Two solutions emerged

denotational semantics

Scott, Strachey
Oxford PRG

$\Downarrow$

LECTURE 2

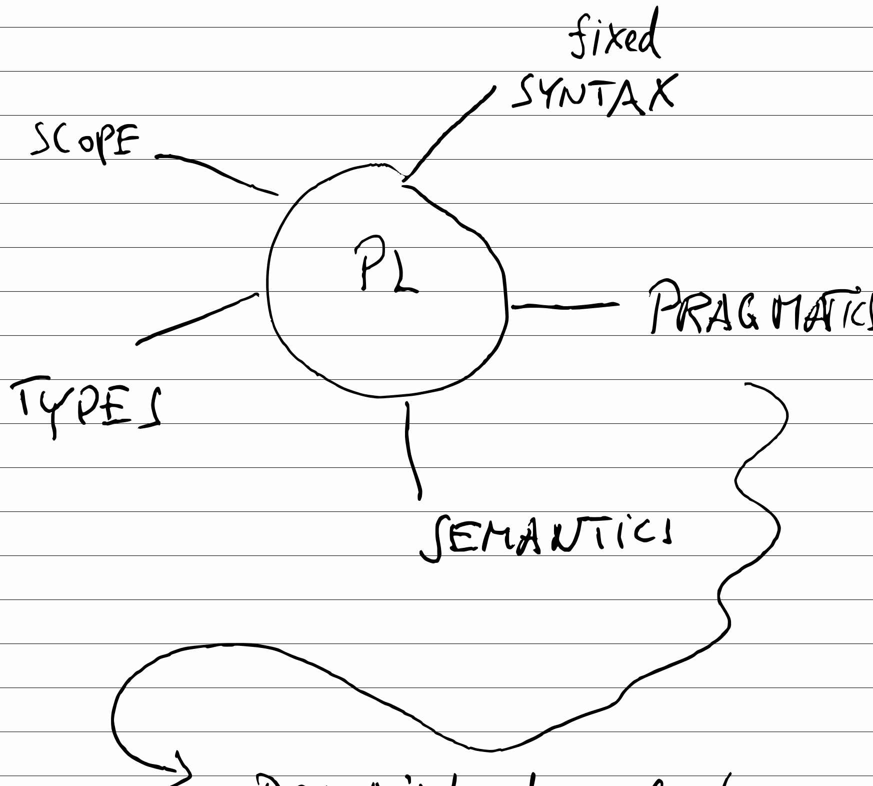
reductions + calculus

Plotkin
Edinburgh

$\Downarrow$

LECTURE 3

THINKING OUTSIDE THE BOX



DOMAIN of application

$\therefore$

MANY DOMAINS

$\therefore$

MANY syntaxes, scopes, ...

from the beginning, some people pursued an alternative:

- extensible syntax
- extensible language
- 2-level language
- language families

Lisp is the survivor of this world:

⟶ Scheme

⟶ Racket

⟶ Clojure

and language workbenches plus projectional editing systems ("intentional programming") are new editions.

Lectures 4 and 5
will introduce this world
a bit:

(1) how do you specify
extension to your
PL ?

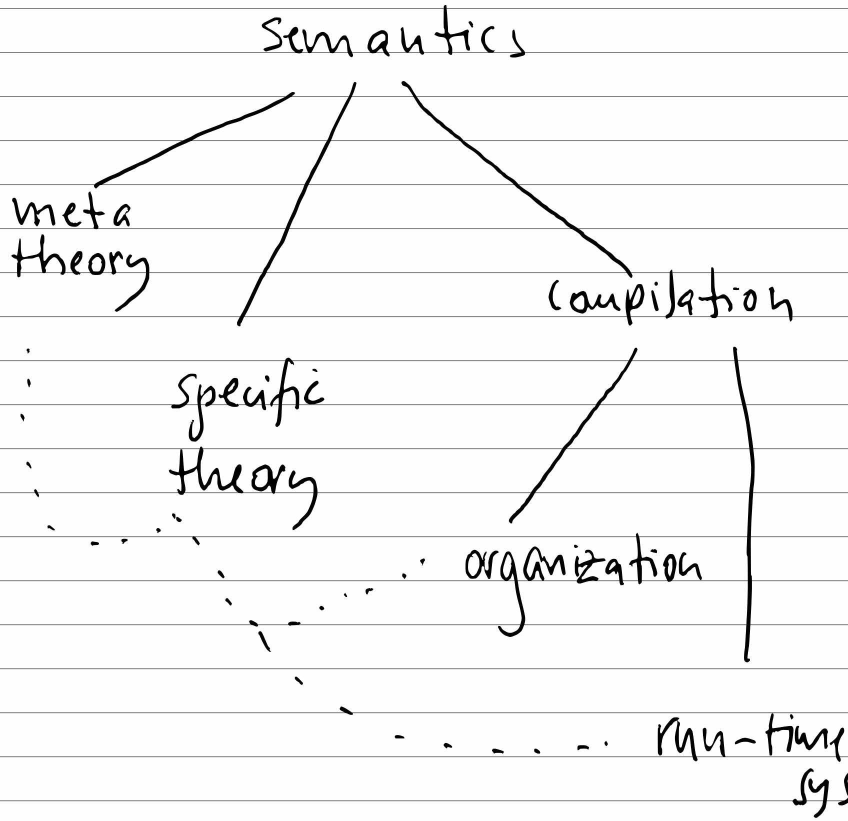
(2) how do macros work
and work w/ the
underlying PL (scope)

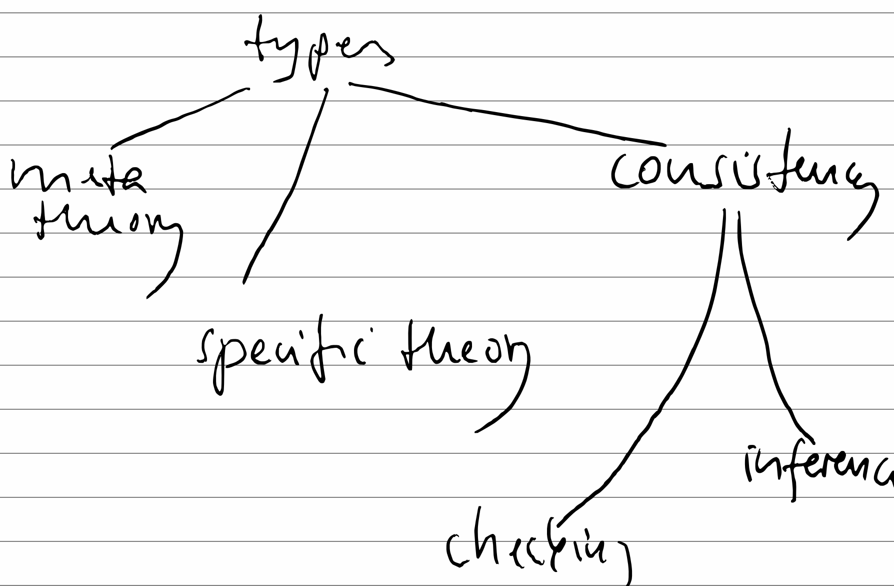
The Course

1. It is about ideas and concepts in PL.

2. Inside the box and outside the box are welcome.

3. Your topic must relate to PL in some way.


semantics
meta theory
specific theory
compilation
organization
run-time sys


types
meta theory
specific theory
consistency
checking
inference

pragmatics

SE

specific features
(control)

analysis
(bug
finding)

verification
(FM)

dealing w/ features
(d/dx)

DSL usage

evaluation

syntax & scope

macros . . . . error reporting
parsers

WHO ARE YOU?

MECHANICS