The Next 700 Programming Languages

Peter J. Landin, 1966

Peter J. Landin

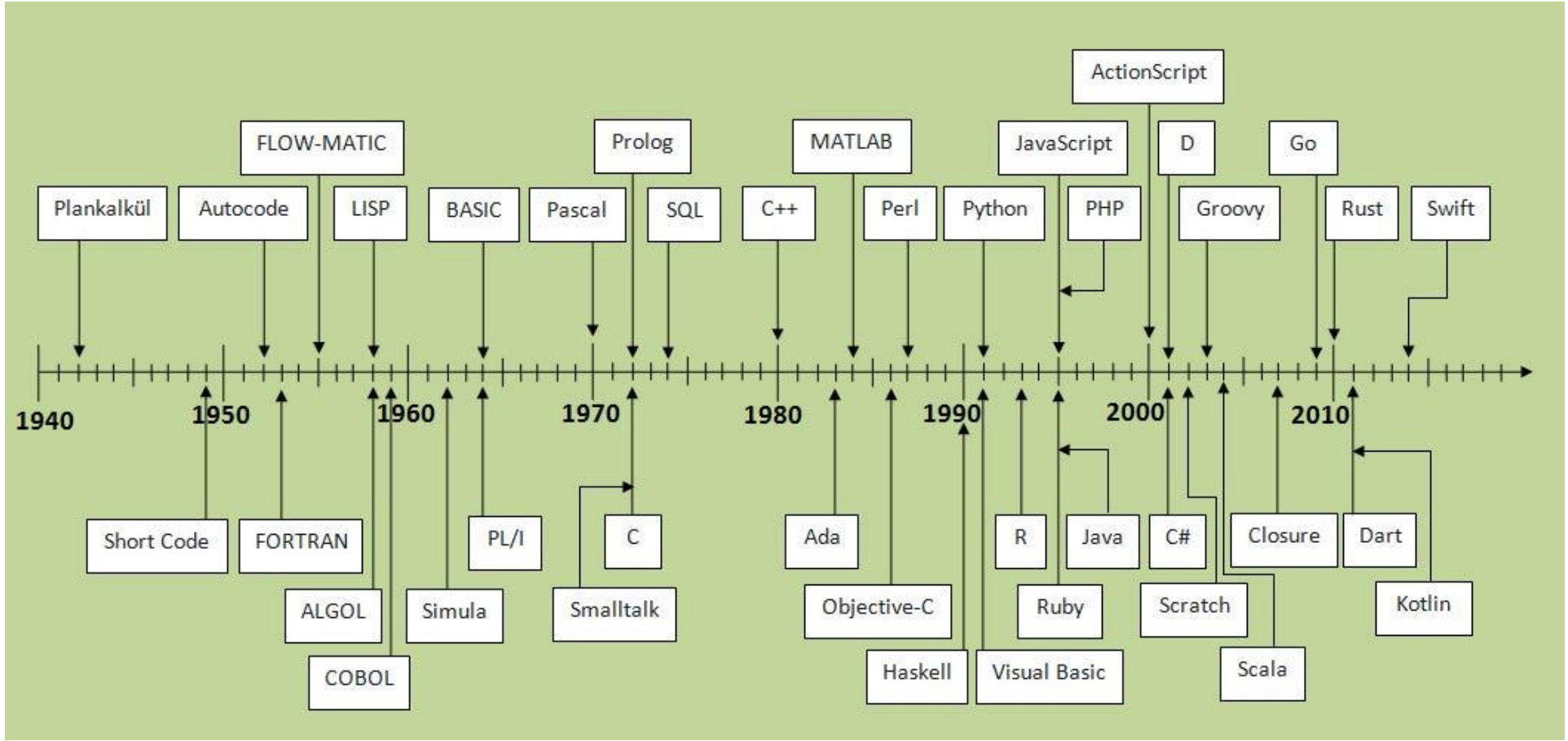
- Britischer Informatiker, 1927–2009
- Einer der Begründer der **funktionalen Programmierung**
- Konzipierung und Entwicklung
  - **ISWIM** (If you See What I Mean)
  - **SECD-Maschine**
  - Mitgewirkt Definition **ALGOL**
- Einfluss auf Sprachen wie **Haskell, ML, Scheme, OCaml**

Motivation des Papers

- 1960er: Explosion neuer Programmiersprachen
- Viele Sprachen sind nur Varianten von Syntax, aber ähnliche Grundideen
- **Landins Ziel:**

  Finde die gemeinsame Basis – die "nächsten 700" Sprachen sollen nur noch Varianten einer klaren Idee sein.
- Diese Basis: **Lambda-Kalkül** + **funktionale Abstraktion**
- Kritik an imperativer Denkweise (Schritt-für-Schritt-Befehle)

Motivation des Papers



ISWIM

- If you See What I Mean
- Eine Sprachfamilie
- Funktionen, Ausdrücke, **where** Klauseln statt Zuweisungen
- Fokus auf **Bedeutung (Semantik)**
- Programme können mit mathematischen Ausdrücken beschrieben werden

Von LISP zu ISWIM

- LISP (1958), erste funktionale Sprache
- Landin bewundert LISP, will aber:
  - weniger Klammern
  - klarere Semantik
  - lesbarere Notation (where, let, Einrückung)
  - **LISP Programme werden Hardware abhängig beschrieben**
- ISWIM = LISP aufgeräumt und verallgemeinert

Mathematisch	**LISP**
1 + (2 * 3) – 4	(- (+ 1 (* 2 3)) 4)

Mathematische Notation

```
f(b + 2c) = f(2b - c)
  where f(x) = x(x + a)
  and b = u / (u + 1)
  and c = v / (v + 1)
```

Definition Where Klausel

1. Linguistic Structure
   - Wo und wie kann die where Klausel verwendet werden
2. Syntax
   - Klare Regel wie where Klausel geschrieben wird (Klammern etc.)
   - **Syntax und Bedeutung trennen**
3. Semantic Contraints
   - Was kann where repräsentieren?
   - integer: where n = round(n)
4. Outcome
   - Was soll passieren?
   - Komplexere Konfigurationen (z.Bsp. verschachtle where)

Physical vs. Logical Language

- Physical Language
  - Syntax, Struktur, effektiver Code
  - Verschiedene Renderings vom abstrakten Programm
- Logical Language
  - Semantik, abstrakte logische Sprache
  - **Reasoning Compiler oder Mathematiker**

Abstraction Layers

1. Physical ISWIM
2. Logical ISWIM
3. Abstract ISWIM
   - Baumstruktur
   - amessages, aexp, adef
4. Applicative Expressions (AEs)
   - Minimaler mathematischer Kern
   - Lambda-Kalkül

Äquivalenzregeln

- Menge formaler Regeln **wann zwei Programme äquivalent** sind.
- **Referential transparency:** Ein Ausdruck kann durch einen gleichwertigen ersetzt werden.

Vier Äquivalenzregeln

Gruppe	Bedeutung	Beispiel
(1) **Subexpression Equivalence**	Wenn zwei Teilausdrücke gleichwertig sind, ist auch der ganze Ausdruck gleichwertig.	`(x+1)*2 ≡ (1+x)*2`

(2) **Definitions**	`let`/`where` können ausgetauscht werden, wenn sie dieselbe Bindung erzeugen.	`let x = a+1 in x*2`
		≡
		`(x*2) where x = a+1`

(3) **Conditionals & Listings**	Bedingungen oder Listen behalten Gleichwertigkeit ihrer Arme.	`if True then a` `else b ≡ a`
(4) **Problem Orientation / Extensions**	Neue Definitionen oder Typen können eingeführt werden.	`f(x)= x+1`

Denotation und Application

- **Denotation**
  - *what an expression means*
  - Jeder Ausdruck definiert ein abstraktes Objekt: eine Zahl, eine Funktion, eine Liste usw.
- **Application**
  - *applying one expression to another*
- Eine **Application** verknüpft zwei Denotationen:
  - Die Denotation einer Funktion
  - Die Denotation ihres Arguments

Denotation und Application

Haskell:

```
f x = x + 1  -- Denotation
f 2          -- Application (den f and den 2)
```

Einfluss auf Haskell

- Haskell beinhaltet Landins Grundideen:
  - reine Funktionen, keine Seiteneffekte
  - where / let Klauseln
  - Rekursion statt Schleifen
  - Äquivalenzregeln (β-Reduktion)

ISWIM vs Haskell

- Haskell

```
f y = x * (x + y)
  where x = a + 2*b
```

- Reference ISWIM

```
f(y) = x(x + y)
  where x = a + 2*b
```

Haskell

1. Haskell Source Code (Physical)

```
sumList :: [Int] -> Int
sumList []     = 0
sumList (x:xs) = x + sumList xs
```

2. Parsed Haskell (Logical)

```
FunctionDecl "sumList"
  TypeSig :: [Int] -> Int
  Equations:
    []     -> 0
    (x:xs) -> x + sumList xs
```

Haskell

3. GHC Core (Abstract)

```
sumList =
  \xs -> case xs of
           []     -> I# 0#
           (x:xs) -> +# x (sumList xs)
```

4. GHC Core bereits nah an Applicative Expressions

Kernideen ISWIM

- Abstract and Physical Language
  - Trennung zwischen Konzept und Darstellung
- Vier Abstraktionsebenen
  - Physical → Logical → Abstract → Applicative Expressions
- Äquivalenzregeln
  - Wann sind zwei Programme gleichwertig?
- Denotation und Application
  - Denotation = meaning or value of an expression
  - Application = applying one expression to another (function call)
- Eliminierung expliziter Sequenzierung
  - Weniger „erst das, dann das“, mehr „so ist es definiert“