On correspondences between programming languages and semantic notations

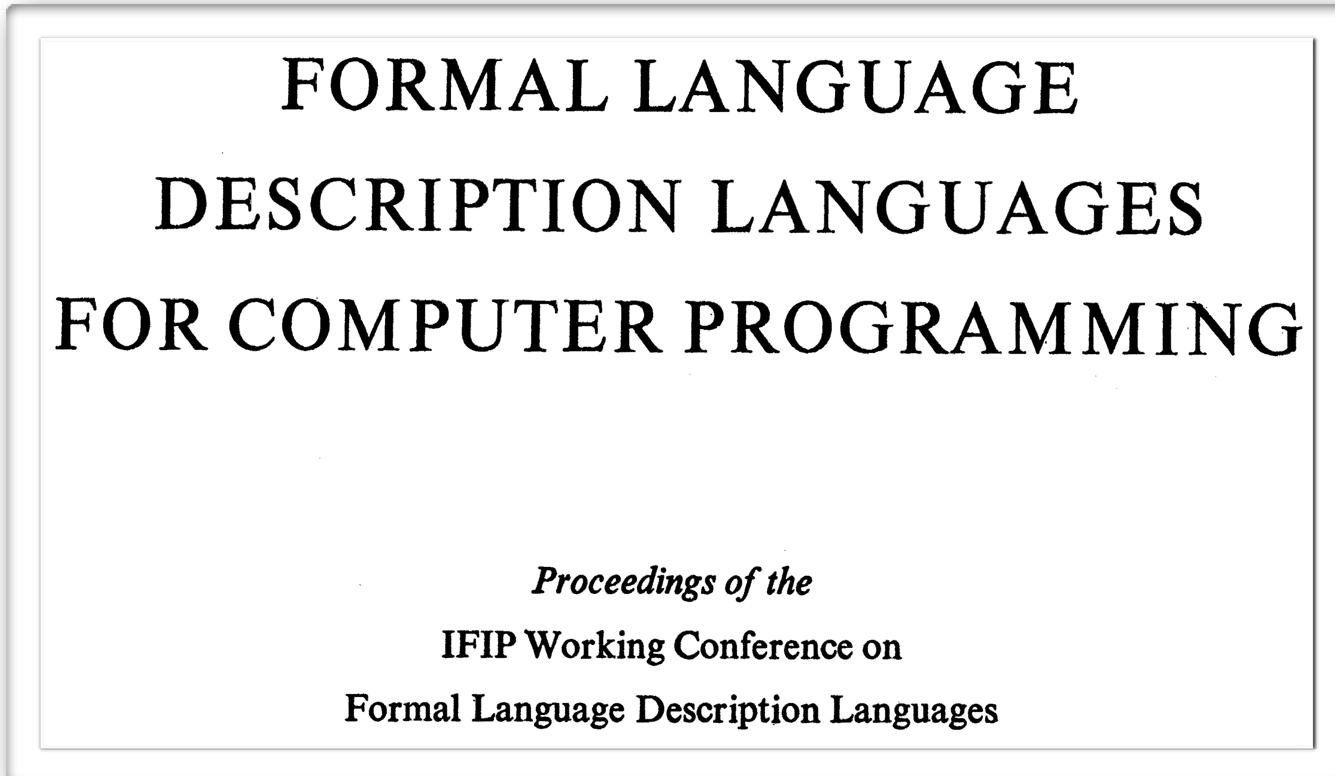
Peter Mosses

Swansea University, UK

BCS-FACS Annual Peter Landin Semantics Seminar

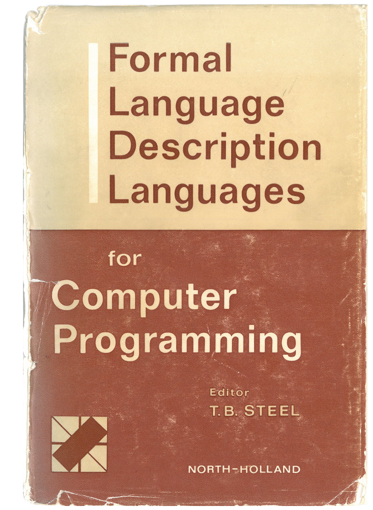
8th December 2014, London

50th anniversary!



FORMAL LANGUAGE
DESCRIPTION LANGUAGES
FOR COMPUTER PROGRAMMING

Proceedings of the
IFIP Working Conference on
Formal Language Description Languages



IFIP TC2 Working Conference, 1964

- 50 invited participants
- seminal papers by ***Landin***, ***Strachey***, and many others
- proceedings published in 1966

Landin and Strachey (1960s)

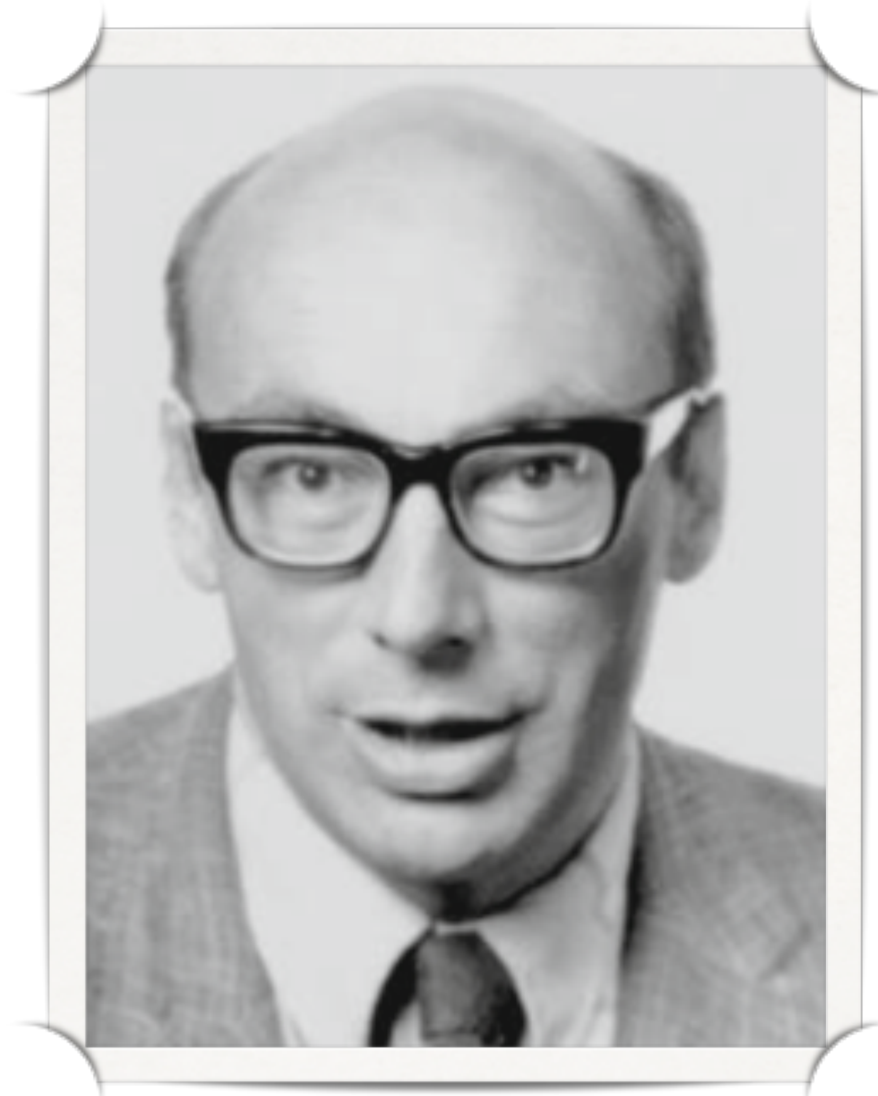
Denotational semantics (1970s)

A current project

Peter J Landin (1930–2009)

Publications 1964–66

- *The mechanical evaluation of expressions*
- *A correspondence between ALGOL 60 and Church's lambda-notation*
- *A formal description of ALGOL 60*
- *A generalization of jumps and labels*
- *The next 700 programming languages*



[http://en.wikipedia.org/wiki/Peter_Landin]

1964

The mechanical evaluation of expressions

By P. J. Landin

This paper is a contribution to the "theory" of the activity of using computers. It shows how some forms of expression used in current programming languages can be modelled in Church's λ-notation, and then describes a way of "interpreting" such expressions. This suggests a method, of analyzing the things computer users write, that applies to many different problem orientations and to different phases of the activity of using a computer. Also a technique is introduced by which the various composite information structures involved can be formally characterized in their essentials, without commitment to specific written or other representations.

Introduction

The point of departure of this paper is the idea of a machine for evaluating schoolroom sums, such as

1. $(3 + 4)(5 + 6)(7 + 8)$
2. **if** $2^{19} < 3^{12}$ **then** $^{12}\sqrt{2}$ **else** $^{53}\sqrt{2}$
3. $\sqrt{\left(\frac{17\, cos\, \pi/17 - \sqrt{(1 - 17\, sin\, \pi/17)}}{17\, cos\, \pi/17 + \sqrt{(1 + 17\, sin\, \pi/17)}}\right)}$

Any experienced computer user knows that his activity scarcely resembles giving a machine a numerical expression and waiting for the answer. He is involved with flow diagrams, with replacement and sequencing, with programs, data and jobs, and with input and output. There are good reasons why current information-processing systems are ill-adapted to doing sums. Nevertheless, the questions arise: Is there any way of extending the notion of "sums" so as to serve some of the needs of computer users without all the elaborations of using computers? Are there features of "sums" that correspond to such characteristically computerish concepts as flow diagrams, jobs, output, etc.?

This paper is an introduction to a current attempt to provide affirmative answers to these questions. It leaves many gaps, gets rather cursory towards the end and, even so, does not take the development very far. It is hoped that further piecemeal reports, putting right these defects, will appear elsewhere.

Expressions

Applicative structure

Many symbolic expressions can be characterized by their "operator/operand" structure. For instance

$$a/(2b + 3)$$

can be characterized as the expression whose operator is '/' and whose two operands are respectively '*a*,' and the expression whose operator is '+' and whose two operands are respectively the expression whose operator is '×' and whose two operands are respectively '2' and '*b*,' and '3.' Operator/operand structure, or "applicative" structure, as it will be called here, can be exhibited more clearly by using a notation in which each operator is written explicitly and prefixed to its operand(s), and each operand (or operand-list) is enclosed in brackets, e.g.

$$/(a, +(\times(2, b), 3)).$$

This notation is a sort of standard notation in which all the expressions in this paper could (with some loss of legibility) be rendered.

The following remarks about applicative structure will be illustrated by examples in which an expression is written in two ways: on the left in some notation whose applicative structure is being discussed, and on the right in a form that displays the applicative structure more explicitly, e.g.

$$\begin{array}{ll} a/(2b + 3) & /(a, +(\times(2, b), 3)) \\ (a + 3)(b - 4) + & +(\times(+(a, 3), -(b, 4)), \\ (c - 5)(d - 6) & \quad \times(+(c, 5), -(d, 6))). \end{array}$$

In both these examples the right-hand version is in the "standard" notation. In most of the illustrations that follow, the right-hand version will not adhere rigorously to the standard notation. The particular point illustrated by each example will be more clearly emphasized if irrelevant features of the left-hand version are carried over in non-standard form. Thus the applicative structure of subscripts is illustrated by

$$a_j b_{jk} \qquad a(j)b(j, k).$$

Some familiar expressions have features that offer several alternative applicative structures, with no obvious criterion by which to choose between them. For example

$$3 + 4 + 5 + 6 \qquad \begin{cases} +(+(+(3, 4), 5), 6) \\ +(3, +(4, +(5, 6))) \\ \Sigma'(3, 4, 5, 6) \end{cases}$$

where Σ' is taken to be a function that operates on a list of numbers and produces their sum. Again

$$a^2 \qquad \begin{cases} \uparrow(a, 2) \\ square\ (a) \end{cases}$$

where ↑ is taken to be exponentiation.

1964

The mechanical evaluation of expressions.
The Computer Journal (1964) 6: 308-320.

- applicative expressions (AEs)
  - *λ-abstraction, application*
- structure definitions
  - *algebraic data types*
- an abstract machine
  - *stack (S), environment (E), control (C), dump (D)*

Applicative expressions (AEs)

An AE is either

an *identifier*,

or a *λ-expression* (*λexp*) and has a *bound variable* (*by*)
which is an identifier or
identifier-list,
and a *λ-body* (*body*)
which is an AE,

or a *combination* and has an *operator* (*rator*)
which is an AE,
and an *operand* (*rand*)
which is an AE.

Applicative expressions (AEs)

- ***AEs*** (*X*) generally ***have values*** (in environments *E*)
  - *independently of any machine*

$$
\begin{aligned}
&\textbf{recursive}\quad valEX = identifierX \rightarrow EX \\
&\qquad\qquad\qquad\qquad \lambda expX \rightarrow f \\
&\qquad\qquad\qquad\qquad\quad \textbf{where}\ fx = val(derive(assoc(bvX,x))E) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (bodyX) \\
&\qquad\qquad\qquad\qquad \textbf{else} \rightarrow \{valE(ratorX)\}[valE(randX)].
\end{aligned}
$$

Syntactic sugar for AEs

- *Lists*

$$x, y, z = x : (y, z) = x : (y : \mathit{unitlist}\ z) = x : (y : (z : ())).$$

- *Conditional expressions*

$$\textbf{if } a = 0 \textbf{ then } 1 \textbf{ else } 1/a \qquad \mathit{if}\,(a = 0)(\lambda().1,\ \lambda().1/a)().$$

- *Recursive definitions, "paradoxical" fixed-point operator (Y)*

$$L = (a, L, (b, c)) \qquad L = Y\lambda L.(a, L, (b, c))$$

$$f(n) = \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } nf(n-1) \qquad f = Y\lambda f.\lambda n.\textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } nf(n-1).$$

1964–65

Standards

S. GORN, Editor; R. W. BEMER, Asst. Editor, Glossary & Terminology
E. LOHSE, Asst. Editor, Information Interchange
R. V. SMITH, Asst. Editor, Programming Languages

A Correspondence Between ALGOL 60 and Church's Lambda-Notation: Part I*

BY P. J. LANDIN†

This paper describes how some of the semantics of ALGOL 60 can be formalized by establishing a correspondence between expressions of ALGOL 60 and expressions in a modified form of Church's λ-notation. First a model for computer languages and computer behavior is described, based on the notions of functional application and functional abstraction, but also having analogues for imperative language features. Then this model is used as an "abstract object language" into which ALGOL 60 is mapped. Many of ALGOL 60's features emerge as particular arrangements of a small number of structural rules, suggesting new classifications and generalizations.

The correspondence is first described informally, mainly by illustrations. The second part of the paper gives a formal description, i.e. an "abstract compiler" into the "abstract object language." This is itself presented in a "purely functional" notation, that is one using only application and abstraction.

Contents

Introduction

Anyone familiar with both Church's λ-calculi (see e.g. [7]) and ALGOL 60 [6] will have noticed a superficial resemblance between the way variables tie up with the λ's in a nest of λ-expressions, and the way identifiers tie up with the headings in a nest of procedures and blocks. Some may also have observed that in, say

$$\{\lambda f.f(a) + f(b)\}[\lambda x.x^2 + px + q]$$

the two λ-expressions, i.e. the *operator* and the *operand*, play roughly the roles of block-body and procedure-declaration, respectively. The present paper explores this resemblance in some detail.

The presentation falls into four sections. The first section, following this introduction, gives some motivation for examining the correspondence. The second section describes an abstract language based on Church's λ-calculi. This abstract language is a development of the AE/SECD system presented in [3] and some acquaintance with that paper (hereinafter referred to as [MEE]) is assumed here. The third section describes informally, mainly by illustrations, a correspondence between expressions of ALGOL 60 and expressions of the abstract language. The last section formalizes this correspondence; it first describes a sort of "abstract ALGOL 60" and then presents two functions that map expressions of abstract ALGOL 60 into, on the one hand, ALGOL 60 texts, and on the other hand expressions of the abstract language.

Motivation

It seems possible that the correspondence might form the basis of a formal description of the semantics of ALGOL 60.[1] As presented here it reduces the problem of specifying ALGOL 60 semantics to that of specifying the semantics of a structurally simpler language. The formal treatment of the latter problem is beyond the scope of this paper, and hence likewise a formal proof that the correspondence described here is correct. It is hoped that the informal account of the semantics of the abstract "object language"

* Part II of this paper, which gives the Formal Presentation of the Correspondence, will appear in the March, 1965 issue of the *Communications of the ACM*.

† Present address: Univac Division of Sperry Rand Corporation, Systems Programming Research, New York, New York.

[1] This view is expanded in [10].

Standards

S. GORN, Editor; R. W. BEMER, Asst. Editor, Glossary & Terminology
E. LOHSE, Asst. Editor, Information Interchange
R. V. SMITH, Asst. Editor, Programming Languages

A Correspondence Between ALGOL 60 and Church's Lambda-Notation: Part II*

BY P. J. LANDIN†

Introduction

The first part of this paper described an abstract language based on Church's λ-calculi, and comprising expressions called "imperative applicative expressions" (IAEs). An informal account was given of the way IAEs can be considered as a generalization and structural simplification of ALGOL 60. The present part presents a formal mapping of ALGOL 60 into IAEs.

Formal Presentation of the Correspondence

The correspondence between ALGOL 60 and IAEs is presented here largely in terms of AEs, or more precisely, in AEs written with an informal syntax that was mostly described in [MEE]. So IAEs figure firstly as analyses of expressions of ALGOL 60 and secondly as description of the analytical process itself. This dual role of IAEs is in a sense fortuitous. However, it has incidental advantages. It provides an example of the use of AEs as a descriptive tool. It also saves us the burden of introducing yet another language in our attempt at language explication.

The presentation that follows somewhat resembles a syntax-oriented compiler in that it is composed of two expressions, namely: first, a syntactic expression that determines a parenthesization of each well-formed ALGOL 60 text and a classification of the parenthesized segments; and, second, a "semantic function" that associates an IAE with each parenthesized text, and hence (on the assumption of unique parenthesization) with each text.

We elaborate the notion of a parenthesized text as follows. We characterize a certain class of constructed objects (*COs*), called *ALGOL 60 COs* (ACOs), which can be considered as abstract ALGOL 60 programs and parts of programs. Each ACO corresponds to an ALGOL 60 text (or more accurately, to a class of texts that are, in a fairly trivial sense, "mutually interchangeable") that can be considered as the written representation of the abstract ACO. Our syntactic expression is an expression denoting a function *sprogram* that passes from an ACO to the class of texts representing it. Our semantic function, *nprogram*, passes from an ACO to an IAE that models it. In choosing an IAE that models a particular ALGOL 60 program there are many decisions to be made, some trivial and some more interesting in that they are analogous to important decisions made when implementing ALGOL 60. However, for our present purpose the main virtues are conciseness and transparency of the semantic function, even at the cost of these qualities in the resulting IAEs.

ABSTRACT ALGOL

The structure definition that follows characterizes a class of constructed objects, called *ALGOL 60 COs* (ACOs), that mirror ALGOL 60 programs. The relation between ALGOL 60 texts and ACOs is many-one; it would be one-one but for spaces, comments, parameter delimiters and the optional omission in ALGOL 60 of **'real'** in certain occurrences of **'real array'.** (It is likely that some other languages make more use of such trivial equivalences.) In framing the definition of ACOs, there is no attempt to filter out just nonsense as

if $a+b$ **then** ...

or

real x; **if** x **then** ...

whereas the syntactic expressions of the ALGOL 60 do exclude the first, if not the second. (Four equivalent identifiers, '*arithexp*', '*Boolexp*', '*designexp*' and '*exp*', are used below solely for improved readability. Their merit depends on our undertaking not to interchange them misleadingly.)

In general, ACOs are more tolerant than ALGOL 60 syntax. In some cases, such as the ones given above, the license is short-lived because it leads to undefined results, but others will actually be given a meaning by our semantic function, e.g.

... **real procedure** a; $a := b+c$;
procedure f; **value** a;...

and

... L: **if** p **then** s; L;...

The structure definition of ACOs uses certain auxiliary

* Received April, 1964; revised November, 1964. Part I of this paper appeared in the *Communications of the ACM 8*, 2 (1965), 89–101. The complete list of References appears with Part I.

† Present address: Univac Division of Sperry Rand Corporation, Systems Programming Research, New York, New York.

1964–65

A correspondence between ALGOL 60 and Church's lambda-notation. *Comm. ACM (1965) 8: 89–101, 158–165.*

- imperative applicative expressions (IAEs)
  - *AEs + program-points (**J**), assigners (lhs ⇐ rhs)*
- correspondence
  - *ALGOL 60 abstract syntax (ACOs)*
  - *synthetic syntax functions : ACOs → Sets(Texts)*
  - *semantic functions : ACOs → IAEs*

ALGOL 60

Landin was an adviser on the official language definition

Revised report on the algorithmic language ALGOL 60

Dedicated to the memory of William Turanski

by

J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy,
P. Naur, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois,
J. H. Wegstein, A. van Wijngaarden, M. Woodger

Edited by

Peter Naur

The report gives a complete defining description of the international algorithmic language ALGOL 60. This is a language suitable for expressing a large class of numerical processes in a form sufficiently concise for direct automatic translation into the language of programmed automatic computers.

The introduction contains an account of the preparatory work leading up to the final conference, where the language was defined. In addition the notions reference language, publication language, and hardware representations are explained.

In the first chapter a survey of the basic constituents and features of the language is given, and the formal notation, by which the syntactic structure is defined, is explained.

The second chapter lists all the basic symbols, and the syntactic units known as *identifiers*, *numbers*, and *strings* are defined. Further, some important notions such as quantity and value are defined.

The third chapter explains the rules for forming expressions, and the meaning of these expressions. Three different types of expressions exist: arithmetic, Boolean (logical), and designational.

The fourth chapter describes the operational units of the language, known as *statements*. The basic statements are: *assignment* statements (evaluation of a formula), *go to* statements (explicit break of the sequence of execution of statements), dummy statements, and *procedure* statements (call for execution of a closed process, defined by a procedure declaration). The formation of more complex structures, having statement character, is explained. These include: *conditional* statements, *for* statements, *compound* statements, and *blocks*.

In the fifth chapter the units known as *declarations*, serving for defining permanent properties of the units entering into a process described in the language, are defined.

The report ends with two detailed examples of the use of the language, and an alphabetic index of definitions.

Contents

ALGOL 60

Landin was an adviser on the official language definition

Revised report on the algorithmic language ALGOL 60

Dedicated to the memory of William Turanski

by

J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy,
P. Naur, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois,
J. H. Wegstein, A. van Wijngaarden, M. Woodger

Edited by

Peter Naur

Algol 60

Expressions

- arithmetic, relational, logical, conditional
- function application, array and switch components

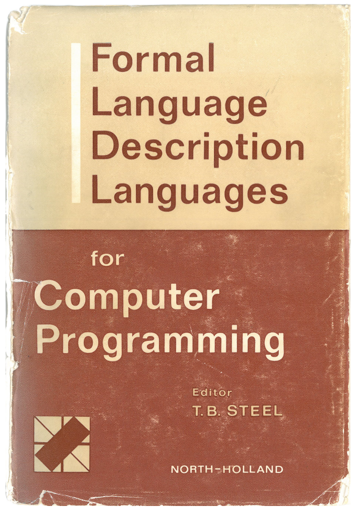
Statements

- assignment, conditional, for-loop, compound
- procedure call, jump

Declarations

- variable, function, procedure, array, switch
- block, recursion, name and value parameters

1964–66



P. J. Landin: ***A formal description of ALGOL 60.***
In Proc. IFIP 1964 TC2 Working Conference on Formal Language Description Languages, 1966.

- an introduction to the full description
- illustration of the correspondence
- discussion of foundations

1965

P. J. Landin: ***A generalization of jumps and labels.***
Univac Technical Report, August 1965;
Higher-Order & Symbolic Computation (1998) 11: 125–143.

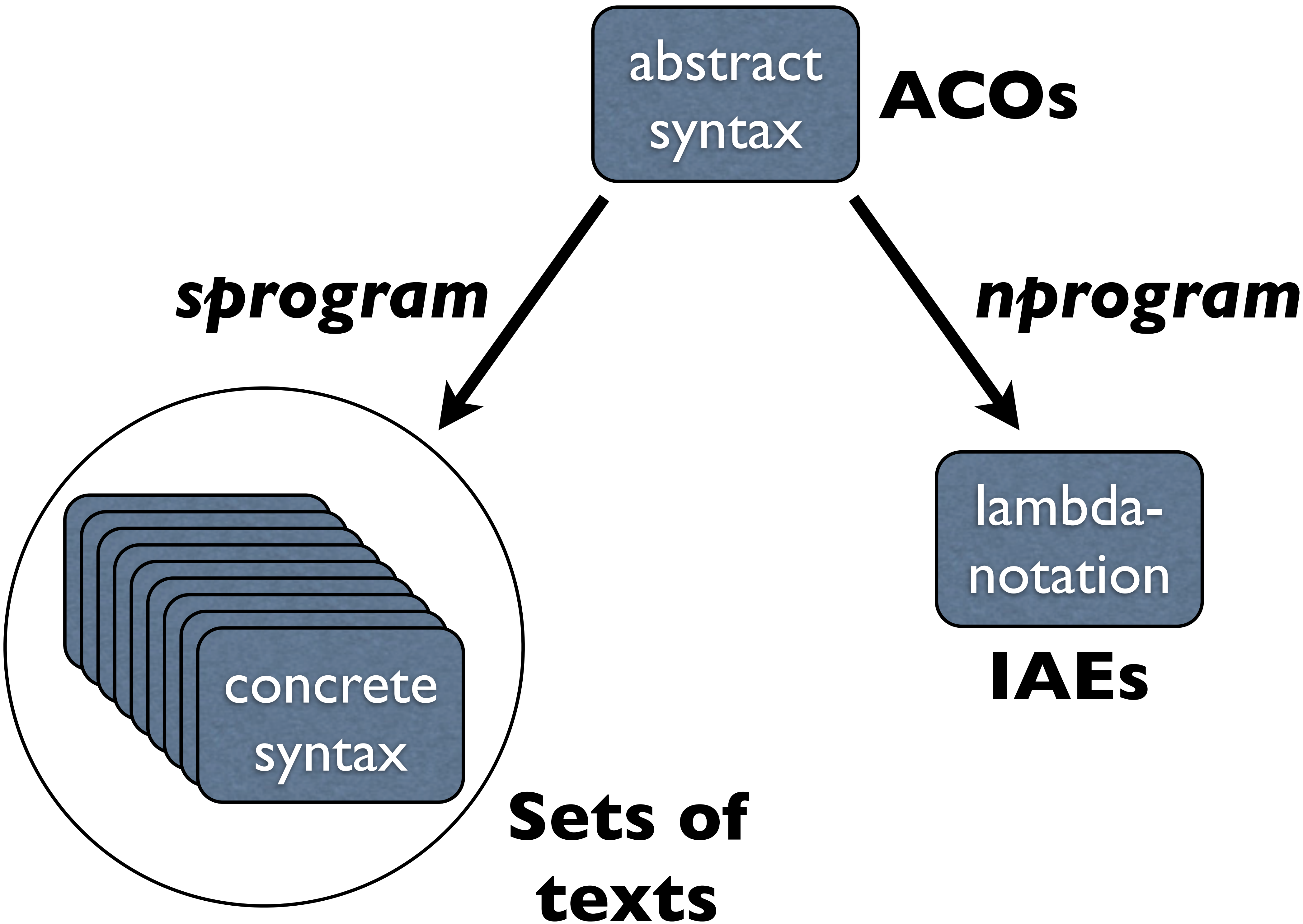
- program-closures, ***J***
  - *syntactic sugar for program-points*
- an extended SECD-*machine*

1965–66

P. J. Landin: ***The next 700 programming languages.***
Presented at an ACM Programming Languages and Pragmatics Conference, California, 1965; Comm. ACM (1966) 9: 157–166.

- **ISWIM** (If you See What I Mean)
  - *family of unimplemented(?) languages*
  - *extends IAEs with (let, where, rec, pp) definitions*

Landin's description of ALGOL 60



Abstract syntax (ACOs)

"presented informally as a rather long English sentence…"

A program is a labeled closedprogram,
where rec
a closedprogram is
either a *block* and has
a *head* which is a nonnull decl-list,
and a *body* which is an openprogram,
or **else** is an openprogram,
where an openprogram is a nonnull labeled(statement)-list,
and a statement is
either *cond* and is
either 2*armed* and has
a *condition* which is a Boolexp,
and a 1*starm* which is a labeled uncondstatement,
and a 2*ndarm* which is a labeled statement,
or 1*armed* and has
a *condition* which is a Boolexp,
and an *arm* which is
either *looping* and is a labeled forstatement,
or **else** is a labeled uncondstatement,
or *looping* and is a forstatement,
or **else** is an uncondstatement,
and a forstatement has
a *control* which is a variable,
and a *forlist* which is a nonnull forlistelement-list,
and a *body* which is a labeled statement,
where a forlistelement is
either a *progression* and has
an *initial* which is an arithexp,
and an *incr* which is an arithexp,
and a *terminal* which is an arithexp,
or an *iteration* and has
a *rhs* which is an arithexp,
and a *condition* which is a Boolexp,
or **else** is an arithexp,
and an uncondstatement is
either *composite* and is a closedprogram,
or *jumping* and has a *body* which is a designexp,
or a *dummy*,
or *assigning* and is an assignexp,
where rec an assignexp has
a *lhs* which is a variable,
and a *rhs* which is
either *simple* and is an exp,
or **else** is an assignexp,
or **else** is a functiondesig,
and a labeled(S) is
either *tagged* and has
a *label* which is an identifier,
and a *body* which is a labeled(S),
or **else** is an S.
and a decl is either *nonrec* and is a nonrecdecl,
or *rec* and is a recdecl,
where a nonrecdecl has
an *ownness* which is a truth-value,
and a *body* which is
either a *typedecl* and has
a *type* which is a classexp,
and a *nee* which is a nonnull identifier-list,
or an *arraydecl* and has
a *type* which is a classexp,
and a *body* which is a nonnull arraysegment-list,
where an arraysegment has
a *nee* which is a nonnull identifier-list,
and a *size* which is a (2-arithexp-list)-list,
and a recdecl is
either a *switchdecl* and has
a *nee* which is an identifier,
and a *niens* which is a nonnull designexp-list,
or a *procdecl* and has
a *type* which is a classexp,
and a *nee* which is an identifier,
and *formals* which are a (possibly null) identifier-list,
and a *valpart* which is a (possibly null) identifier-list,
and a *specpart* which is a (possibly null) spec-list,
where a spec has
a *specifier* which is a classexp,
and a *body* which is a nonnull identifier-list,
and a *body* which is either *code*,
or **else** is a labeled statement,
where rec
an exp is either *cond* and has
a *condition* which is a Boolexp,
and a 1*starm* which is a simpexp,
and a 2*ndarm* which is an exp,
where a simpexp is a
2*op*('≡',
2*op*('⊃',
2*op*('∨',
2*op*('∧',
1*op*('¬',
2*op*('<' | '≦' | '=' | '≧' | '>' | '≠',
2*op*('+' | '−',
1*op*('+' | '−',
2*op*('×' | '/' | '÷',
2*op*('↑', typeprimary)))))))))
where a 1*op*(o,S) is
either 1*compound* and has
a *rator* which is an o,
and a *rand* which is an S,
or **else** is an S,
and a 2*op*(o,S) is
either 2*compound* and has
a *rator* which is an o,
and a 1*strand* which is a 2*op*(o,S)
and a 2*ndrand* which is an S,
or **else** is an S,
and a typeprimary is
either a *const* which is
either *arithmetical*,
or *Boolean*,
or *string*,
or *simple* and is a variable,
or **else** is an exp,
and an arithexp is an exp,
and a Boolexp is an exp,
and a designexp is an exp,
and a classexp is
either *simple* and is **'real'** | **'integer'** | **'Boolean'** | **'string'** |
'label' | **'command'**,
or **else** has a *rator* which is **'array'** | **'procedure'**,
and a *rand* which is a classexp,
and a variable is
either *simple* and is an identifier,
or an *element* and has
a *rator* which is an identifier,
and a *rand* which is a nonnull arithexp-list,
or **else** is a functiondesig,
and a functiondesig has
a *rator* which is an identifier,
and a *rand* which is a nonnull exp-list.

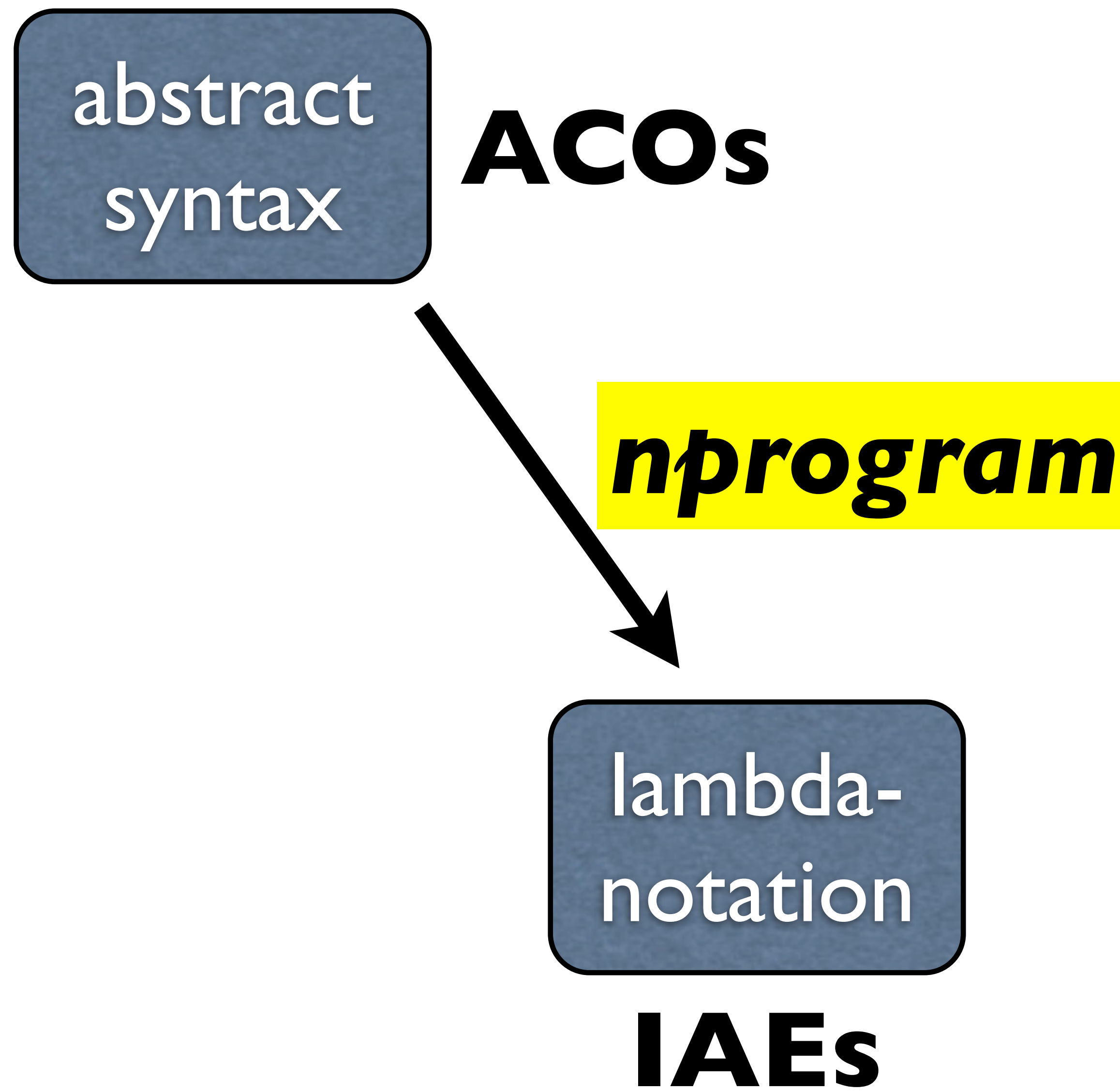
Volume 8 / Number 3 / March, 1965
Communications of the ACM **159**

Abstract syntax (ACOs)

"presented informally as a rather long English sentence…"

and an uncondstatement is
either *composite* and is a closedprogram,
or *jumping* and has a *body* which is a designexp,
or a *dummy*,
or *assigning* and is an assignexp,
where rec an assignexp has
a *lhs* which is a variable,
and a *rhs* which is
either *simple* and is an exp,
or **else** is an assignexp,
or **else** is a functiondesig,
and a labeled(S) is
either *tagged* and has
a *label* which is an identifier,
and a *body* which is a labeled(S),
or **else** is an S.

Landin's description of ALGOL 60



nprogram : ACOs → IAEs

is merely an abbreviation for

```
combine ('f', conslisting (combine('g', 'x'), 'y'))
```

We start with two list-processing functions, that are needed later:

```
rec map f L = null L → ( )
              else → f(hL) : mapf(tL)
e.g.  map square (1, 3, 7, 2, 3) = (1, 9, 49, 4, 9)

unzip L = (map 1st L, map 2nd L)
e.g.  unzip ((1,2), (3,4), (5,6), (7,8)) = ((1,3,5,7), (2,4,6,8))

rec select(p)(L) = null L → ()
                   p(hL) → hL:select(p) (tL)
                   else → select(p) (tL)
```

Next come functions for processing IAEs and definitions. These are built up from the constructors for IAEs, namely *consλexp*, *conspp*, *consassigner* and *combine*. They also use the selectors and constructor for definitions, namely *nee*, *niens* and *consdef*.

```
combine2(F,X,Y) = combine(combine(F,X),Y)
e.g.  combine2("D","sin","kπ/2") = "{Dsin}[kπ/2]"

combine3(F,X,Y,Z) = combine(combine2(F,X,Y),Z)

combinelist(F,X) = combine(F,conslisting(X))
e.g.  combinelist("f",("a+b","c+d")) = "f(a+b,c+d)"

consβredex(J,X,Z) = combine(consλexp(J,X),Z)
e.g.  consβredex("u","u(a+u)","b+c") = "{λu.u(a+u)}[b+c]"
  i.e.  "u(a+u) where u = b+c"

consYredex(J,X) = combine('Y', consλexp(J,X))
e.g.  consYredex("L","a:(b:L)") = "YλL.a:(b:L)"
  i.e.  (roughly) "L where rec L = a:(b:L)"

delay(X) = consλexp(conslisting( ),X)
e.g.  delay("f(a) + f(b)") = "λ( ).f(a) + f(b)"

do(X) = combinelist(X,( ))
e.g.  do("x") = "x( )"

conscondexp(P,X) = do(combinelist(combine('if',P), map delay X))
e.g.  conscondexp("a=0","a","1/a") =
        "if(a=0)(λ( ).a, λ( ).1/a)( )"

delayed(F) = combine('B', F)
e.g.  delayed("float") = "Bfloat"
  i.e.  (roughly) "f where f(x) = float·x"

serial(X) = delay(serial '(X,conslisting( )))
  where serial' (X,Z) = nullX → Z
    else → serial'(tX,combine(hX,Z))
e.g.  serial("R", "S", "T") = "λ( ).T(S(R( )))"
  i.e.  "T·S·R"

parallel(D) = let J = map nee D
              and Z = map niens D
              consdef(conslistingJ, conslistingZ)
e.g. parallel ("let u = a+b",
               "let v = c+d",
               "let w = e+f") =
               "let (u, v, w) = (a+b, c+d, e+f)"

conscondexp'(P, F) = consλexp('x', conscondexp(do P, map do F))
                       where do(f) = combine(f, 'x')
e.g.  conscondexp'("P ∪ q", "f", "g·h") =
                     "λx.(P ∪ q) (x) → f(x)
                         else → (g·h) (x)"

jump (X) = combine ('J', X)
```



```
consletexp(D, X) = consβredex(nee D, X, niens D)
e.g.  consletexp ("let x = 2p − q", "x(x + 1)") =
                  "let x = 2p − q
                   x(x + 1)"

consrecexp(D,X) = consβredex(nee D, X, consYredex
                                        (nee D, niens D))
e.g.  consrecexp ("let x = x²+¼", "x(x+1)")=
                  "let rec x = x²+¼
                   x(x+1)"

labels (N, X) = consβredex(N, X, map jump N)
e.g.  labels ("L, M", "φ") =
              "let L = JL
               and M = JM
               φ"

arrangeaspseudoblock(D, X)  =  conrecexp(D, labels(nee D, X))

arrangeasblock(D, D', D'', X) =
  consletexp(D, consrecexp(parallel(D',D''), labels(nee D'', X)))
e.g.  (roughly)
arrangeasblock("let a = φ1", "letp(x)=φ2",
                       "letL=φ4 and M=φ5", "φ3")=
  "let a = φ1
   let rec p(x) = φ2
       and L = φ4
       and M = φ5
   let L = JL
   and M = JM
   φ3"

arrangeasblock'(D, D'', X) = arrangeasblock(D, parallel(), D'',X)
```

There now follows the definition of the semantic function *nprogram*, with its auxiliaries *nforlistelement*, *nexp*, etc.

```
nprogramS = arrangeasblock'(nlabeled(nclosedprogramN0'I')S)
where rec
  nclosedprogramNLS =
    blockS → let D0, D, D', X0 =
                 nhead(N, derive(classifiedvariablesS)N) (headS)
             let D0'', D'', X'' = nopenprogram(pnovN)L(bodyS)
             (parallel(D0, D0''),
              parallel( ),
              arrangeasblock(D,D', D'', serial(X0,X'')))
    else → nopenprogramNLS
    where nopenprogramNLS =
      null(tS) → nlabeled(nstatementNL)(hS)
      else → let j, N', N'' = takenewlabelN
             let D0, D, X = nlabeled(nstatementN'j)(hS)
             let D0', D', X' = nopenprogram(pnovN'')L(tS)
             (parallel(D0, D0'),
              parallel(D, consdef(j, delayX'), D'),
              X)
and nstatementNLS =
  condS →
    let D0, D, X =
      2armedS → nlabeled(nuncondstatementNL)(1starmS)
      1armedS → nlabeled(looping(armS) → nforstatementN
                         else → nuncondstatementNL)
                        (armS)
    let D0', D', X' =
      2armed → nlabeled(nstatementNL)(2ndarmS)
      1armed → parallel( ), parallel( ), 'I'
    (parallel(D0, D0'),
     parallel(D, D'),
     conscondexp(nBoolexp(conditionS), (X, X')))
  loopingS → serial(nforstatementNS, L)
  else → nuncondstatementNLS
```



```
and nforstatementNS =
  let D0, D, X = nlabeled(nstatementN'I')(bodyS)
  (D0,
   parallel( ),
   combinelist('for',
               (nlhsN(controlS),
                combinelist('concatenate*',
                            map(nforlistelementN)(forlistS)),
               arrangeaspseudoblock(D, X)))
  where nforlistelementNS =
    progressionS →
      combinelist('step*',
                  map(narithexpN)
                      (initialS, incrS, terminalS))
    iterationS →
      combinelist('while*',
                  (narithexpN(rhsS),
                  nBoolexpN(conditionS)))
    else → combine('unitlist*', narithexpNS)
and nuncondstatementNLS =
  compositeS → nclosedprogramNLS
  else → (parallel( ),
          parallel( ),
          (jumpingS → ndesignexpN(bodyS)
           else →
             serial((dummyS → 'I'
                     assigningS → combine2('K', conslisting( ),
                                                 nassignexpNS)
                     else → nfunctiondesigNS),
                    (L))))
    where rec nassignexpNS =
      combine2('assignandhold',
               (simpleS → nexpNS
                else → nassignexpN(rhsS)),
               nlhsN(lhsS))
and nlabeled(ncategory)(S) =
  taggedS → let D0, D, X = nlabeled(ncategory)(bodyS)
            D0, parallel(consdef(labelS, delayX), D), labelS
  else → ncategoryS
and nhead(N, N')(S) =
  let D0', D' = map parallel
                    (unzip(map nrecdeclN'(select rec S)))
  (parallel(D0': map nnonrecdeclN(select ownness S)),
   parallel(map nnonrecdeclN(select(¬·ownness) S)),
   D',
  serial(map nownarraydeclresetN (select(arraydecl∪ownness) S)))
  where nnonrecdeclNS = typedeclS → ntypedeclNS
                        arraydecl S → narraydeclNS
    and nrecdeclNS = switchdecl S → nswitchdeclNS
                     procdecl S → nprocdeclNS
  where ntypedeclNS =
    parallel(map ntypedecl'(neeS))
    where ntypedecl'J =
      consdef((ownnessS → ownvariantNJ
               else → J),
              combine('separate',
                      initialcon(typeS)))
  and narraydeclNS =
    let J, Z = unzip(map narraysegmentN(bodyS))
           where narraysegmentNS' =
      conslisting(ownnessS → map ownvariantN
                                             (neeS')
                  else → nee S'),
      consβredex('A',
                 conslisting(map(K"separate A")
                                 (nee S'))
                 combine2('expandtoarray',
                          nbplistN(sizeS),
                          'x'))
```



```
    consdef(conslisting J,
            consβredex('x',
                       conslisting Z,
                       initialcon(typeS)))
  and nownarraydeclresetNS =
      serial(map nownarraysegmentreset(bodyS))
      where nownarraysegmentresetNS' =
        serial(map nownarrayreset(neeS'))
        where nownarrayreset J =
          consassigner(ownvariantNJ,
                       combine2 ('parearray',
                                 nbplist(sizeS'),
                                 conslisting(ownvariantNJ,
                                             initialcon(typeS))))
      where nbplistNS' =
        conslisting(map nboundpairNS')
        where nboundpairNS' =
          conslisting(ownnessS → ('−∞', '+∞')
                      else → map narithexpNS')
  and nswitchdeclNS =
    consdef( ),
    consdef(neeS,
      combine2('arrangeasarray',
               conslisting(u(conslisting('1',
                                         length(niensS)))),
               conslisting(map ndesignexpN(niensS))))
  and nprocdeclNS =
    let D0'', D'', X'' =
      code(bodyS) → ncode(derive(classifiedvariablesS)N)
                         (bodyS)
      else → nlabeled(nstatement(derive(classifiedvariablesS)N)
                                ('I')(bodyS)
    D0'',
    consdef(neeS,
            consλexp(conslisting(formalsS),
                     (typeS = 'command' → X
                      else → consβredex(resultvariant(neeS),
                                        X,
                                        initialcon(typeS)))
      where X = consβredex(J,
                           arrangeasblock'(parallel(map nspecN
                                                        (specpartS)),
                                           D'',
                                           X''),
                           Z)
    where nvalue J = consdef(J, combine('separate', doJ))
    and nspecNS = parallel(map nspec'N(bodyS))
      where nspec'NJ =
        consdef(J,
                combine((needsapplyingNJ → delayed
                         else → I)(transfer)
                                 (specifierS)
                                 (J))
where rec
  nexp = ncond(nsimp(ntypeprimary))
  where ncond(ncategory)NS =
    condS → conscondexp(nBoolexpN(conditionS),
                        (ncategoryN(1starmS),
                        ncond(ncategory)N(2ndarmS)))
  and nsimp(nprimary)NS =
    1compoundS → combine(monadicvariant(ratorS),
                         nsimp(nprimary)(randS))
    2compoundS → combinelist(ratorS,
                             map(nsimp(nprimary))
                                (1strandS, 2ndrandS))
  and ntypeprimaryNS = constS → S
                       simpleS → nvariableNS
                       else → nexpS
```



```
and narithexp = nexp
and nBoolexp = nexp
and ndesignexp = nexp
and nvariableNS = simpleS → nidentifierNS
                  elementS →
                    combine(nidentifierN(ratorS),
                      conslisting(map narithexpN(randS)))
                  else → nfunctiondesigNS
and nlhsNS = simpleS ∧ putativeresultNS → resultvariantS
             else → nvariableS
and nfunctiondesigNS =
  combinelist(nidentifier(ratorS),
              map(delay·nexp)(randS))
where nidentifierNJ = needsapplyingNJ → doJ
                      ownidentifierNJ → ownvariantNJ
                      else → J
and initialcon A = A = 'real' → '0.0'
                   A = 'integer' → '0'
                   A = 'Boolean' → 'false'
and transfer S = simple S → S = 'real' → 'float'
                            S = 'integer' → 'unfloat'
                            S = 'command' → combine
                                              ('in', 'null')
                            S = 'label' → 'I'
                            else → combine('in', S)
                 else →
                   conscondexp'('atom',
                                (t, combine('B', t)))
                   where t = transform(rand S)
where needapplying N = needsapplying N∪
                           putativeresultandneedsapplying N
and putativeresult N = putativeresult N∪
                           putativeresultandneedsapplying N
and monadicvariant J = (J = '+') → '+M'
                       (J = '−') → '−M'
```

nprogram : ACOs → IAEs

where rec $nassignexpNS =$

$\quad combine_2(\text{‘}assignandhold\text{’},$

$\qquad (simpleS \rightarrow nexpNS$

$\qquad\quad \textbf{else} \rightarrow nassignexpN(rhsS)),$

$\qquad nlhsN(lhsS))$

$assignandhold(x)(y) = \textbf{let } x = \textbf{real } y \rightarrow float\ x$

$\qquad \textbf{integer } y \rightarrow unfloat\ x$

$\qquad \textbf{Boolean } y \rightarrow in(\textbf{Boolean})x$

$\quad 2nd((y \Leftarrow x), x)$

Landin’s description of ALGOL 60

Virtues

- a ***major example*** of a correspondence between a ***real*** programming language and a semantic notation
  - *concisely documented Landin’s expert analysis*
  - *demonstrated the use of AEs (ISWIM) as a meta-language*

Drawbacks

- correspondence not tested/validated
  - *no tool support (?)*
- sharing of addresses not defined
- fixed-point operator ***Y*** defined in terms of assigners

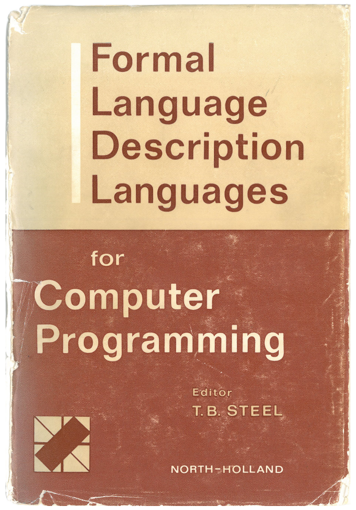
Landin and **Strachey (1960s)**

Denotational semantics (1970s)

A current project

Christopher Strachey (1916–1975)

Towards a formal semantics. *In Proc. IFIP 1964 Working Conf. on Formal Language Description Languages, 1966.*



- cited by Landin as ***an alternative to IAEs***
  - *"to find, for each command, an AE denoting the SECD-transformation it effects"*
- introduces ***L-values*** and ***R-values***
  - *an L-value "denotes an area of the **store**"*
- refers to the fixed-point operator ***Y*** as "paradoxical"
  - *cites Landin's "computing procedure" for it*

Assignments

- *Without side-effects:*

$$\boxed{\begin{aligned} Prog \quad \{\epsilon_1 := \epsilon_2\} &= \lambda\sigma.\, U\alpha_1(\beta_2, \sigma) \\ \text{where } \alpha_1 &= L(\epsilon_1, \sigma) \\ \text{and } \beta_2 &= R(\epsilon_2, \sigma) \end{aligned}}$$

- *With side-effects:*

$$\boxed{Prog\{\epsilon_1 := \epsilon_2\} = \lambda\sigma\,.\,(\lambda(\alpha, \sigma')\,.\,U\alpha(R'(\epsilon_2, \sigma')))\,(L'(\epsilon_1, \sigma))}$$

Strachey's 1960s approach

Virtues

- used to give a correspondence between a ***developing*** major programming language (CPL) and a semantic notation
- applicative definition of addresses, stores, assigners
  - *avoided the need for an abstract machine*

Drawbacks

- meta-language left informal
- fixed-point operator ***Y*** left "paradoxical"

Landin and Strachey (1960s)

Denotational semantics (1970s)

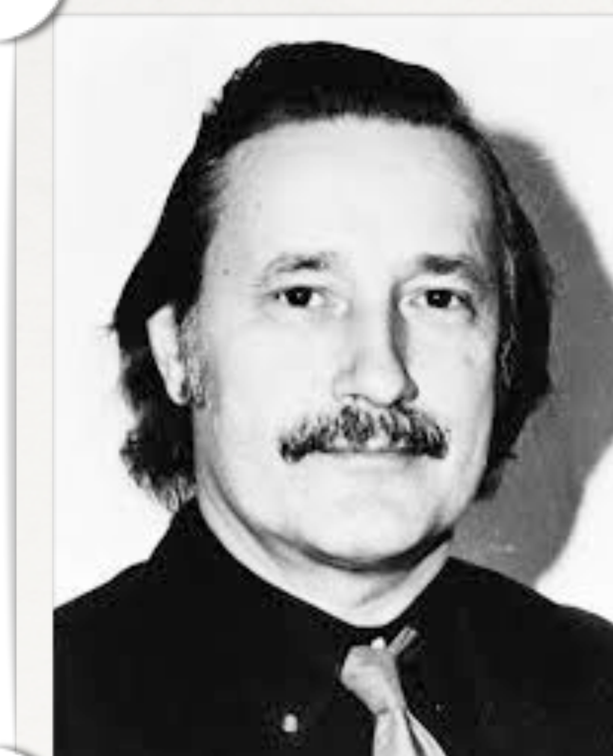
A current project

Autumn 1969

Dana Scott: ***Some reflections on Strachey and his work.***

Higher-Order and Symb. Comp. (2000) 13: 103–114

- "The semester at Oxford […] was a very intense time, working with ***Strachey***, meeting ***Peter Landin***, ***David Park***, ***John Reynolds***, and many others. As I recounted in my Turing Lecture, I did not intend at all to ***construct models for the type-free λ-calculus***…"



Scott–Strachey semantics

D. S. Scott, C. Strachey: ***Towards a mathematical semantics for computer languages.***

Technical Monograph PRG-6, Oxford Univ. Comp. Lab, 1971

- least-fixed point operator ***Y*** no longer 'paradoxical'
- correspondence between program phrases and their ***denotations*** in Scott-domains (originally *lattices*, later *cpos)*
- separation between ***environments*** $\rho \in Env$ and ***stores*** $\sigma \in S$
  - $\mathcal{C} : Cmd \to (Env \to (S \to S))$
  - $\mathcal{E} : Exp \to (S \to (T \times S))$

Scott–Strachey semantics

D. S. Scott, C. Strachey: ***Towards a mathematical semantics for computer languages.***

Technical Monograph PRG-6, Oxford Univ. Comp. Lab, 1971

- ***Abbreviations:***
  - $(f \circ g)(\sigma) = f(\sigma')$ when $g(\sigma) = \sigma'$
  - $(f * g)(\sigma) = f(\beta)(\sigma')$ when $g(\sigma) = (\beta, \sigma')$
  - $(P\ \beta)(\sigma) = (\beta, \sigma)$

monadic notation!

- $\mathcal{C}[\![\gamma_0 \,;\, \gamma_1]\!] = \lambda\rho.\ \mathcal{C}[\![\gamma_1]\!](\rho) \circ \mathcal{C}[\![\gamma_0]\!](\rho)$
- $\mathcal{C}[\![\varepsilon \rightarrow \gamma_0 \,,\, \gamma_1]\!] = \lambda\rho.\, \mathit{Cond}(\mathcal{C}[\![\gamma_0]\!](\rho), \mathcal{C}[\![\gamma_1]\!](\rho)) * \mathcal{E}[\![\varepsilon]\!]$

Scott–Strachey semantics

PDM: ***The mathematical semantics of* A*LGOL* 60.**

Technical Monograph PRG-12, Oxford Univ. Comp. Lab, 1974

- ***Continuations-style***
  - $\mathcal{C} : \mathit{Sta} \rightarrow (\mathit{Env} \rightarrow (C \rightarrow C))$ where $C = (S \rightarrow S)$
  - $\mathcal{R} : \mathit{Exp} \rightarrow (\mathit{Env} \rightarrow (X \rightarrow (K \rightarrow C)))$ where $K = (V \rightarrow C)$
  - $\gamma_1 \mid\mid \gamma_2 \mid\mid \theta = \gamma_1\{\gamma_2\{\theta\}\}$
- $\mathcal{C}[\![\mathit{Sta}\ \mathbf{;}\ \mathit{StaL}]\!] = \lambda\rho.\,\lambda\theta.\ \mathcal{C}[\![\mathit{Sta}]\!]\rho \mid\mid \mathcal{C}[\![\mathit{StaL}]\!]\rho \mid\mid \theta$
- $\mathcal{C}[\![\mathbf{if}\ \mathit{Exp}\ \mathbf{then}\ \mathit{Sta}_1\ \mathbf{else}\ \mathit{Sta}_2]\!] = \lambda\rho.\,\lambda\theta.$

  $\mathcal{R}[\![\mathit{Exp}]\!]\rho\ \text{“boolean”}\ \{\ \lambda\beta.\ \beta \rightarrow \mathcal{C}[\![\mathit{Sta}_1]\!]\rho\theta, \mathcal{C}[\![\mathit{Sta}_2]\!]\rho\theta\ \}$

Scott–Strachey semantics

PDM: ***The mathematical semantics of ALGOL 60.***

Technical Monograph PRG-12, Oxford Univ. Comp. Lab, 1974

- ***Continuations-style***

```
def 𝕮*⟦t:StaL⟧ρθ = switch labelof t in
§
case"Sta ; StaL": 𝕮⟦Sta⟧ρ ‖ 𝕮*⟦StaL⟧ρ ‖ θ
case"Sta":        𝕮⟦Sta⟧ρθ
§
```

```
case"if Exp then Sta₁ else Sta₂":
    ℛ⟦Exp⟧ρ"boolean" {λβ. β → 𝕮⟦Sta₁⟧ρθ, 𝕮⟦Sta₂⟧ρθ}
```

VDM semantics

H. Bekić, D. Bjørner, W. Henhapl, C. B. Jones, P. Lucas:
A formal definition of a PL/I subset.
Tech. Rep. TR 25.139, IBM Lab. Vienna, Dec. 1974

- ***Combinators:*** abbreviations with ***fixed behaviour*** (definitions dependent on the domains of denotations)

```
def id:e; s
```

```
for i=m to n do S(i)
```

```
return(v)
```

```
if t then c else a
```

```
f;g
```

monadic notation!

```
v := e
```

```
c v
```

VDM semantics

W. Henhapl, C. B. Jones: ***A formal definition of ALGOL 60.***

In "The Vienna Development Method: The Meta-Language", LNCS 61: 305–336, 1978; and Chapter 6 of "Formal Specification & Software Development", Prentice-Hall, 1982

- ```
  M:   Stmt → ENV =>
  ```
- ```
  M:   Expr → ENV => VAL
  ```
- ```
  M[mk-Compound(<s1,s2>)](env) =
     M[s1](env); M[s2](env)
  ```
- ```
  M[mk-If(e,th,el)](env) =
     def b: M[e](env);
     if b then M[th](env) else M[el](env)
  ```

VDM semantics

W. Henhapl, C. B. Jones: ***A formal definition of ALGOL 60.***
In "The Vienna Development Method: The Meta-Language", LNCS 61: 305–336, 1978; and Chapter 6 of "Formal Specification & Software Development", Prentice-Hall, 1982

```
M:   Unlabstmt → STMTENV   =>

M:   Expr → EXPRENV => VAL
```

```
M[mk-Compstmt(stl)](stenv) Δ
    for i=1 to lenstl do M[s-sp(stl[i])](stenv)
```

```
M[mk-Condstmt(e,th,el)](stmtenv) Δ
    let (,env,cas) = stmtenv in
    def b              : M[e](env,cas);
    if b then M[s-sp(th)](stmtenv) else M[s-sp(el)](stmtenv)
```

Further reading

PDM: ***VDM semantics of programming languages: combinators and monads.***
Formal Aspects of Computing (2011) 23: 221–238

C. B. Jones: ***Semantic descriptions library***
homepages.cs.ncl.ac.uk/cliff.jones/semantics-library/

- ***searchable on-line resources*** new!
  - *descriptions of ALGOL 60 in various frameworks*
- ***scanned manuscripts***
  - *VDM descriptions of programming languages*
  - *VDL description of PL/I*
  - *…*

CBJ's contribution to PDM "Semantics Library" · Cliff B Jones

homepages.cs.ncl.ac.uk/cliff.jones/semantics-library/

Semantic descriptions library

These are my (current, evolving) contributions to the "library of semantics" being developed in collaboration with the PLanCompS project.

Searchable on-line resources

Thanks to painstaking work by Roberta Velykiene, the following scanned PDFs have an overlay which makes searching possible (even for Greek letters!)

- *Peter Lauer's VDL description of ALGOL 60* (TR 25.088)
- *A "functional" semantics of ALGOL 60* (Notice that this scanned version deliberately omits the pages that contained the ALGOL report that were lined-up with the corresponding formulae)
- *Peter Mosses' (Oxford) Denotational descriptionof ALGOL 60*
- *A (the second) VDM description of ALGOL 60*
- *A re-LaTeXed version of the ALGOL 60 report*

Scanned manuscripts

Landin and Strachey (1960s)

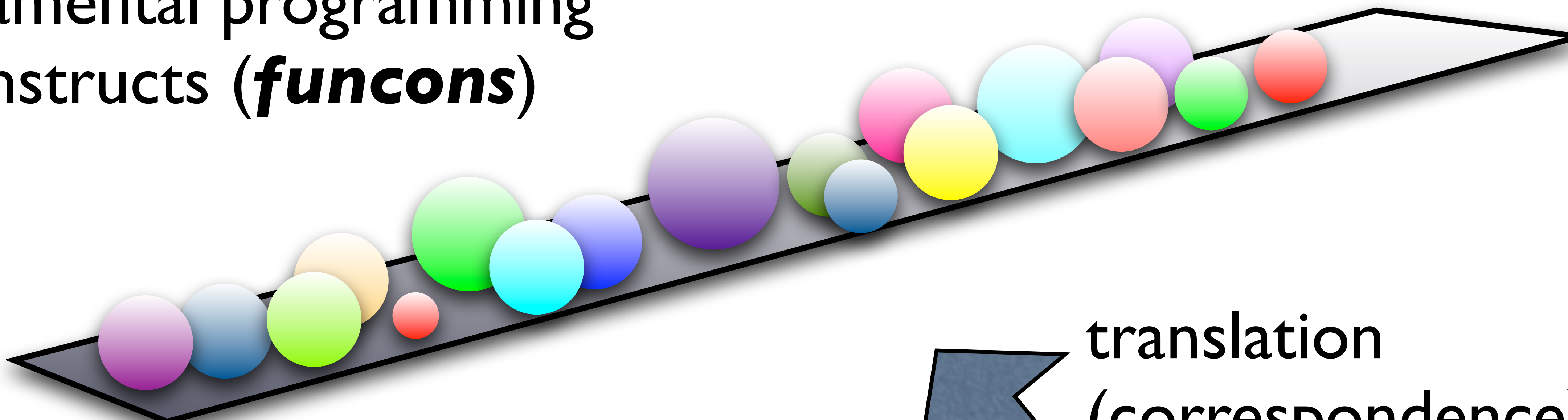
Denotational semantics (1970s)

A current project

Component-based semantics

fundamental programming constructs (***funcons***)

open-ended collection



components-off-the-shelf

translation (correspondence)

evolving languages

Component-based semantics

```
Semantics
  execute[[ _:(statement ( ';' statement )*) ]] : commands

Rule
  execute[[ S1 ';' S2 ...]] =
    sequential(execute[[S1]], execute[[S2 ...]])

Rule
  execute[[ 'if' E 'then' S1 'else' S2 ]] =
    if-true(evaluate[[E]], execute[[S1]], execute[[S2]])
```

Reusable components

Fundamental constructs (funcons)

- correspond to programming constructs
  - ***directly*** (**sequential**, **scope**, …)
  - ***special case*** (**if-true**, **apply**, **assign**,…)
  - ***implicit*** (**bound-value**, …)
- and have *(when validated and released)*
  - ***fixed*** notation
  - ***fixed*** behaviour
  - ***fixed*** algebraic properties

specified/proved once and for all!

PLANCOMPS project (2011-2015)

Foundations

- component-based semantics, bisimulation [Swansea]
- GLL parsing, disambiguation [RHUL]

Case studies

- CAML LIGHT, C#, JAVA [Swansea]

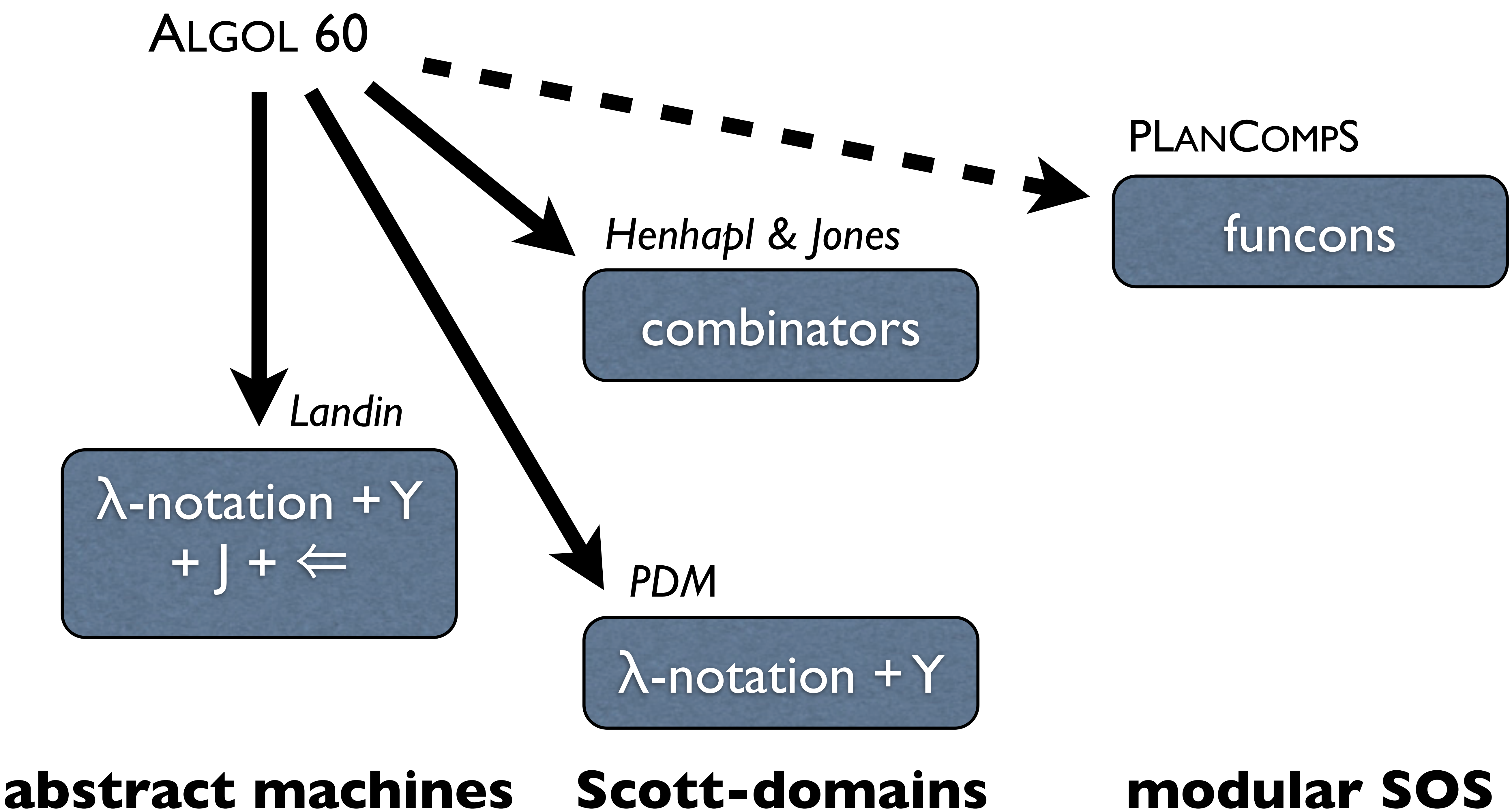
Tool support

- IDE, funcon interpreter/compiler [RHUL, Swansea]

Digital library

- interface [City], historic documents [Newcastle]

Summary



foundations