Landin's Thesis and the Issue of 'Syntactic Sugar'

- The λ-calculus (LC) is said to have laid the foundations not just of Functional Programming (FP) but also of programming in general, despite the Turing-Machine (TM) model being a clearer model of a computing machine. A TM actually proceeds in steps of computation. It can be programmed by a machine table. The Programmable Register Machine (PRAM) improves on that in the model of (random) memory access and comes close to the von Neumann architecture of standard computing machines. Low level programming with Assembler (like) instruction sets takes only a small step from a simple PRAM to higher level programming. A paradigmatic higher level programming language like **C** takes then only a smaller step from Assembler and embodies the paradigm of Imperative Programming (IP) and computer memory management. This procedural approach seems distinct from the λ-calculus.

- In the 1960s/1970s programming was just burgeoning into a manifold of programming languages, **LISP** and **ALGOL** being two of the first programming languages with **LISP** being more functional than imperative and **ALGOL** being more imperative (leading to its proper imperative followers **PASCAL** and **C**), Peter Landin proposed that programming languages are just LC with added 'syntactic sugar' for easier comprehension and practical implementation.[1] This has been dubbed[2]

(*Landin's Thesis*) **Programming Languages are λ-calculus sweetened with syntactic sugar.**

- λ-terms in LC and FP define functions, they give no instructions. Computation in FP occurs in evaluating them, starting from the β-reduction rule (applying the term in the scope to some arguments) and evaluating the resulting terms with respect to the involved function(s). Thus, for instance, in evaluating "λx(x + 1)4" β-reduction yields "4 + 1", and this is evaluated as an addition. Addition has to have been implemented somewhere as a procedure. The static

[1] Cf. Landin, "A correspondence between ALGOL 60 and Church's Lambda-notation".
[2] Cf. Trakhtenbrot, "Comparing the Church and Turing Approaches: Two Prophetical Messages".

impression λ-terms convey hides recurring on implemented (basic) procedures to evaluate λ-terms. In this way Declarative Programming (say, as Logical Programming in **PROLOG**) and Functional Programming (say, in **HASKELL**) rely on the procedures (i.e. Imperative Programming) of their (virtual) evaluation machines. The procedural semantics of a **PROLOG**-program assigns to the program the (unification involving) steps of the resolution algorithm to resolve the program (in contrast to a denotational semantics of the program). Resolution is a procedure. As machine procedures form the core of machine computation *the core* of machine computation is IP. As FP and IP are often contrasted as paradigms this contrast is justified with respect to the style of actual programming languages and corresponding code (say, in **HASKELL** or **PROLOG** in contrast to **C**). Nonetheless – especially given the paradigm contrast between IP and FP – procedures (i.e. IP) seem to be more than just 'syntactic sugar'.

- In LC one can have **if then else**-like statements by an encoding that mimics the branching behaviour; e.g. if the Boolean value **true** is encoded "λxy.x", the value **false** as 'λxy.y", and branching **if then else** as "λbxy.bxy", then by β-reduction **if true t u → t**, because this is "(λbxy.bxy)(λxy.x)t u", accordingly **if false t u → u**. Thus, a code with branching can be expressed within LC. Numeral constants ('Church Numerals') can be introduced by λ-terms with iterated self-application. As also arithmetic operations can be introduced a branching code with arithmetic operations on numerals can be encoded into a – long and hard to read – λ-expression. LC can be understood as a programming language.
Translating such a λ-expression in 'ordinary' code with symbols "+", "if" etc. one can see the LC as the backbone of such symbolic code and the use of the easier to read symbols as 'syntactic sugar'.

- Combining the last two points we can say, on the one hand, that symbolic code is 'syntactic sugar' for λ-expressions, but as λ-expressions are evaluated by implemented procedures which ultimately come down in a machine to Assembler operations, on the other hand symbolic code is closer related to what the machine does. (This is almost tautological for low level programming.)

- Programming languages that implement LC inherit meta-logical features of LC:

(i) They can implement programs that do not return, because to be Turing-complete they have to implement the partial recursive functions.

(ii) Whether a program in such a language terminates or is correct (i.e. computes what it should compute) is not decidable in general.

(iii) The consistency of programs in general in such a language cannot be checked by a program of that language.

Programming languages that enforce termination of every program can do so only at the cost of sometimes giving the wrong answer, and there will be (even) *total* functions which they cannot compute.

These properties correspond to the *Halting Problem (Church's Theorem)*, *Rice's Theorem*, *Gödel's 2nd Incompleteness Theorem*, and the non-enumerability (by diagonalization) of the total recursive functions.