```
tteral) ~(cls:clause list) : clause list =
un (c:clause) ->
t.filter_map (fun (l':litteral) ->
 if l = l' then raise SAT
 else if l = -l' then None
 else Some l') c)
one) cls

(cls:clause list) : litteral =
 (fun (c:clause) -> List.length c = 1) cls)
```

```
let sat ~(cls:clause list) : int
 let rec aux (cls:clause list)
  if cls = [] then Some i
  else if List.mem [] cls then
  else try
     let l = unit_litteral cl
     aux (simplify l cls) (l:
   with Not_found ->
      let l = List.hd (Lis
      match aux (simplify
      | Some _ as res -> r
      | None          -> a
in
aux cls []
```

The Science of Programming

Languages & tools

Parsing of programming languages was based on the study of grammars, formal languages and automata. At ICALP'72, 30 out of 50 presentations dealt with formal languages and automata theory. In the 1970's, the theory of programming languages turned to the description of their semantics with algebra, denotational semantics, and mathematical logic. Since then, new conferences have appeared about logic in computer science, principles of programming languages, compilers, functional programming, types, static analysis, concurrency, automatic verification.

PROGRAMMING LANGUAGES

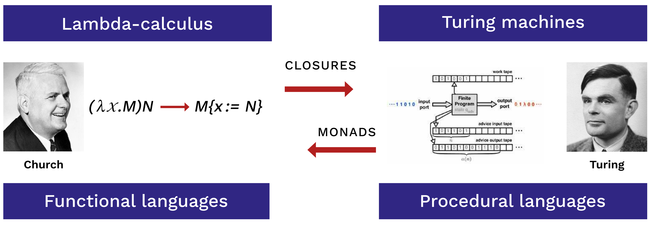
The next 700 programming languages predicted by Peter Landin in 1965 are now nearly existing. Today languages are introduced with their semantics written in a more or less formal setting. Mathematical models have also influenced the design of new concepts (types, closures, objects, etc).

FROM THE 1930'S FOUNDATIONAL COMPUTATION MODELS...

...TO REAL-WORLD LANGUAGES

WITH PROGRAMMING PARADIGMS

- Object-oriented
- Logic
- Scripting
- Descriptive
- Functional
- Procedural

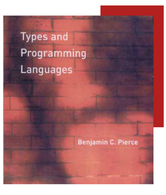


PROGRAMMING TOOLS

The first programming tools dealt with compiler construction or program profiling. Nowadays they include program verification, static analysis, and program testing. These new tools have followed theoretical progress in the semantics of programming languages, dependent high-order types, interactive proof-checkers, automatic provers, and abstract interpretation.

TYPES

POLYMORPHISM
OVERLOADING
GRADUAL TYPING
MODULES AND FUNCTORS



COMPILERS

VIRTUAL MACHINES
OPTIMIZED CODE
SPECIAL HARDWARE

VERIFICATION

LOGIC FOR PROGRAMS
MACHINE-CHECKED PROOFS
STATIC ANALYSIS

CONCURRENCY

RACE-FREE
RESOURCE ALLOCATION
PROOFS