Introduction to the Literature on Programming Language Design

Gary T. Leavens
Iowa State University

Follow this and additional works at: http://lib.dr.iastate.edu/cs_techreports
Part of the Other Computer Sciences Commons, and the Programming Languages and Compilers Commons

Recommended Citation
http://lib.dr.iastate.edu/cs_techreports/99

This Article is brought to you for free and open access by Digital Repository @ Iowa State University. It has been accepted for inclusion in Computer Science Technical Reports by an authorized administrator of Digital Repository @ Iowa State University. For more information, please contact hinefuku@iastate.edu.
Introduction to the Literature
On Programming Language Design

Gary T. Leavens

TR #33-01b

Keywords: programming languages, semantics, type systems, polymorphism, type theory, data abstraction, functional programming, object-oriented programming, logic programming, declarative programming, parallel and distributed programming languages.


© Gary T. Leavens. Permission is granted for you to make copies for educational and scholarly purposes, but not for direct commercial advantage, provided this notice appears on all copies. All other rights reserved.

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1040, USA
Introduction to the Literature
On Programming Language Design

Gary T. Leavens*
Department of Computer Science, Iowa State University
Ames, IA, 50011-1040 USA
leavens@cs.iastate.edu

February 5, 1996

Abstract

This is an introduction to the literature on programming language design and related topics. It is intended to cite the most important work, and to provide a place for students to start a literature search.

This is a selective introduction to the literature on programming language design. The intended audience is graduate students beginning a study of programming languages. Instead of trying to be comprehensive, references are given that are to works of lasting value, up-to-date surveys, or that seem to be important or interesting at the moment. Besides references that have intrinsic interest, a few are included because they are the original sources and are likely to be referenced by others doing related work (e.g., [Chu41]). To probe an area more deeply, start with the papers mentioned, follow their references, and also use the Science Citation Index to see what papers have referenced the ones mentioned.

As a general aid to finding papers, many older references are reprinted in a collection called TUTORIAL Programming Language Design edited by A. I. Wasserman and available from the IEEE Computer Society [Was80]. Readers with web browsers may also want to look at the following web page, titled “Programming Language and Compiler Bibliographies.”

http://www.cs.cmu.edu/~mleone/language/bibliographies.html

However, this tends to have better coverage of recent technical reports, and its coverage of older journal papers and other published material is spotty.

1 Generalities

There are several excellent undergraduate texts on programming languages, including: [PZ96] [Seb96] [Set96] [FWH92] [Hen90] [Kam90] [Wat90] [GJ87] [Mac87] [Ten81]. Graduate texts include: [SK95] [Sta95] [Sch94] [Win93] [Wat91] [Mey90] [Gor88].

Volume B of the Handbook of Theoretical Computer Science contains many detailed surveys relevant to formal models and semantics [vL90].

Journals include a substantial coverage of programming languages include ACM Transactions on Programming Languages and Systems (abbreviated TOPLAS), ACM Letters on Programming Languages and Systems (abbreviated LOPLAS), ACM SIGPLAN Notices, Computer Languages (Pergamon Press), Journal of Programming Languages (Chapman and Hall), Information and Computation (formerly Information and Control Academic Press), and Acta Informatica (Springer-Verlag). Applied areas, such as specification and verification have their own journals (e.g., IEEE Transactions on Software Engineering, Science of Computer Programming, Formal Aspects of Computing). Other journals can be found by scanning the bibliography.

*This work was supported in part by the National Science Foundation under Grants CCR-9108654 and CCR-9303168.

1 Naturally these judgements are personal, but it is hoped that authors of that vast bulk of papers that are not cited here will not take offense.
Conferences devoted to topics related to programming languages include the Annual ACM Symposium on Principles of Programming Languages (POPL), the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), the ACM SIGPLAN International Conference on Functional Programming (formerly called the Symposium on LISP and Functional Programming, LFP), and the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP). Conferences with a more theoretical bent include the IEEE Annual Symposium on Logic in Computer Science (LICS), Mathematical Foundations of Programming Semantics (MFPS), the European Symposium on Programming (ESOP), the International Colloquium on Automata, Languages, and Programming (ICALP), and the International Symposium on Mathematical Foundations of Computer Science. Conferences not sponsored by the ACM or IEEE are often available in the Springer-Verlag series of Lecture Notes in Computer Science (LNCS). The above list does not include conferences devoted to software engineering (e.g., TAPSOFT) or particular methods (e.g., category theory) where other important work is found.

Both Hoare and Wirth have written guidelines for programming language designers [Hoa80] [Wir74]. Floyd’s Turing Award lecture discusses his view of programming languages as capturing paradigms of programming [Flo79]. Another viewpoint is that language design should proceed from semantic principles [Ten77] [Bac78a].

The truism that bad programs can be written in any programming language is described as “Floyd’s Axiom” in a paper written at the height of the structured programming controversy [Flo75]. Shaw and Wulf point out that language designers can provide reasonable defaults while still giving programmers the ability to change them [SW80].

The idea of the expressiveness of a language is just starting to receive satisfactory formal treatments [Fel90] [Mit91].

2 Semantics

Some texts that cover many different approaches to semantics are: [NN92] [Win93] [SK95].

2.1 Operational Semantics

Landin’s SECD machine is defined in his paper “The Mechanical Evaluation of Expressions” [Lan64] (see also Henderson’s book [Hen80]).

A more systematic style of operational semantics based on rewrite rules is found in Plotkin’s structural operational semantics [Plu77] [Hen90] [Aas91].

Several meta-circular interpreters for variants of LISP are discussed in Steele and Sussman’s paper The Art of the Interpreter [SS78a]. An excellent and more readily accessible discussion is found in Abelson and Sussman’s book [ASS85], which uses Scheme. A more detailed treatment of interpreters is found in [Kam90] [FWH92]. See [KdRB91] for an approach using the CLOS meta-object protocol. More recently, structuring mechanisms for semantics based on monads [Mog90] [Wad92] have lead to more modular structuring of definitional interpreters [Ste94] [LH95].

2.2 \(\lambda\)-Calculus

A standard reference on Church’s \(\lambda\)-calculus [Chu41] is Barendregt’s book [Bar84]. Informal introductions include [Gor88]. For the typed \(\lambda\)-calculus, see [GLT89] or [Mit90b].

The use of \(\lambda\)-calculus for describing programming languages and as the inspiration for programming language design has been investigated by Landin [Lan63] [Lan66].

2.3 Denotational Semantics

A short summary of the denotational approach to programming language semantics [Sco81] can be found in Tennenbaum’s article “The Denotational Semantics of Programming Languages” [Ten76]. A survey is found in [Mos90].

Introductory texts include [Gor79], [All86], [Wat91] [Sch94]. An excellent text with mathematical depth is [Gun92]. Standard works on denotational semantics are the books by Stoy [Sto77] and Schmidt [Sch86].
both of which offer a comprehensive and mathematical treatment. Schmidt’s book [Sch86] can be consulted for references to denotational descriptions of real languages.

To read some of the most technical works, one will need some familiarity with category theory [Lan71] [Go84] [LS91]; the following are good introductions that emphasize semantic applications [Hoa89] [BW90a] [AL91b] [Pie91] [Wal91] [Gun92].

One can use a typed functional programming language, such as Standard ML, to implement a denotational semantics. Two descriptions of this are [Wat86] [MA89].

Action semantics, an offshoot of denotational semantics, is described in [Wat91] and more fully in [Mos92].

2.4 Axiomatic Semantics

An early presentation of axiomatic semantics is Hoare’s paper “An Axiomatic Basis for Computer Programming” [Hoa69].

An example of the use of axiomatic semantics to define a programming language is Hoare and Wirth’s axiomatic definition of Pascal [HW73]. A book on axiomatic semantics slanted towards programming language theory (as opposed to verification) is Hesselink’s monograph [Hes92]. Foundational material that may help in reading this monograph is found in [DS90], which goes over notational issues as well as the underlying mathematics.

A very introductory tutorial on verification from the software engineering perspective is [LG86, chapter 11]. The idea of developing a proof of a program at the same time the program is being developed has been eloquently advocated by Dijkstra and Gries [Dij76] [Gri81]. More recent treatments in this style advocate a calculational approach [Mor94] [Coh90] [GS94]. See [Al91a] for an introduction that discusses concurrent and distributed programs.

A survey of program verification for imperative programs is [Cou90].

A language designed to support program verification is Euclid [LGH78] [PHL77]. Another such language is Alphard [SWL77] [Sha81].

2.5 Algebraic Semantics

Another approach to the semantics of programming languages is the algebraic approach; specifying the behavior of programs instead of directly specifying the function computed. A book-length treatment is [BHK95], which uses tools described in [Kli93]. Examples include [MA86] [BWP87] [PGM90] [SK95, Chapter 12].

3 Type Systems

3.1 Background

The purpose of type checking is nicely summarized by Morris as a mechanism that allows program modules to protect objects from unwanted discovery, modification, and impersonation [Mor73]. Wegbreit’s discussion of the extensible language EL1, is also good background [Weg74].

3.2 Polymorphism

An introductory survey of modern polymorphic type systems and research results is Cardelli and Wegner’s paper “On Understanding Types, Data Abstraction and Polymorphism” [CW85]. See also [Har84] [Car91] [DT88]; the latter two have much material related to object-oriented programming. A still more recent survey is [Mit90b].

Standard references include Girard’s system $F\omega$ [Gir71] (see also [Gir86] [GLT89]), and Reynolds independent work [Rey74], sometimes called the Girard-Reynolds second order lambda calculus (or SOL). Modern expositions are found in [MP85] [MH88] [Rey85] [Mit90b] [Car91] [Sch94].

Other kinds of type information may be incorporated into a type system and checked at the same time as types [GL86] [LG88] [OG89].
3.3 Type Inference

Type inference is also sometimes called type reconstruction. The basic idea is described in Milner's paper "A Theory of Type Polymorphism in Programming" [Mil78]. A more modern exposition of Milner's ideas is [Car87].

This type inference system is used in the programming language Standard ML [GMW79] [MTH90] [MT91]. Textbooks about programming in Standard ML include [Pan91] [Sok91] [Sta92] [UI94]. An extension of this type system to handle static overloading (ad hoc polymorphism) is described in [WB99] and further refined in [Jon95] [NP94].

The language Poly [Mat85b] [Mat85a] uses type inference to infer type parameters. An algorithm for inferring operation parameters is described in [CW90]. Other extensions to the basic type inference algorithm are described in [OG89] [Boe89]. Along similar lines, one can use a mix of type annotations and inference to allow polymorphic functions to be passed as arguments [OL96].

The computational complexity of various type checking and type inference problems has also been an active area of research. See [Tiu90] for a survey.

3.4 Type Theory

Type theory, narrowly defined, uses the tools of constructive logic to study type systems such as the above. Logical inference systems can often be translated directly into type systems due to the "Formula as Types" notion or the "Curry-Howard isomorphism" [How80] [GLT93, Chapter 3] [Con89]. Thus much research in type theory lies on the border of mathematics and computer science. Another motivation is to use type information to capture behavioral specifications, thus allowing one to reason about programs in the programming language [NP83] [Dyb90].

After reading the references above, one will still need an introduction to some of the more technical aspects of type theory. Good book-length treatments are [Tho91] and [Sch94]. Those wishing a shorter introduction might try [Rey85] (which is not comprehensive, but is tutorial) [Bac89] (which especially focuses on Martin-Löf style type theory) and [Sce90] (which focuses more on the calculus of constructions). After reading one of these the student may want to read [PDM99] for some practical hints.

The principal groups that have been working on type theory include deBruijn and others working on AUTOMATH [dB80], Martin-Löf's and followers [ML75] [ML82] [Bac89] [BCMS89], Constable's group at Cornell [CZ84], and Coquand and Huet's group [CH88] has also been influential for modern language design.

Some of the latest work on type theory uses linear logic [Gir90] instead of a more standard logic.

3.5 Data Abstraction and Types

A good introductory treatment of the idea of data abstraction is found in [LG86]. A more technically oriented introduction is [CL90]. Another excellent paper is [Coo91], which distinguishes between programming with abstract data types and object-oriented programming.

CLU is a language designed around data abstraction [LSAS77] [LAB81] [LG86]. It also has an interesting control abstraction and exception handling mechanisms [LS79].

Alphard, designed around the same time as CLU and with many of the same goals, has surprising differences [SWL77] [Sha81].

The language Russell was developed at Cornell to investigate how types can be treated as values. There are many papers that have appeared about Russell, but perhaps the best introduction to the language is the paper "Data Types are Values" [DD85], which can be consulted for other references.

Much recent work involves object-oriented languages. For example, Trellis/Owl [SCB+86] features strong type checking and a declared (i.e., by-name) subtype relation. By contrast many other languages feature structural subtyping, including Emerald [BHJL86] [BHJ87] [BH90] [BH91] [Car91]. Theoretical work in this area includes the following [Car86] [Car87a] [AC93] [CMMS94] [CCH94] [CHC90] [Coo90] [BTGS90] [BCM93] [BM92] [Bru93] [BCM93] [PS94] [Bru94] [PT94] [Aba94] [AC94] [AC95] [EST95]. (Cardelli is one of the most active in this area, and most of the literature will cite one of his papers.) For work that directly bears on multimethods (as in CLOS), see [Rey86] [Ghe89a] [Ghe91b] [GCL92] [Cha92] [CGL95] [Cas93] [Cas95] [CL95]. For a tutorial discussion of the problems of typing binary methods, see [BCC+95].
4 Alternative Programming Models

4.1 Functional Programming

A survey of functional programming is [Hud89], which also discusses the language Haskell [HF92] [HJW92] [Sno92] [Dav92]. Another survey is [Bar90]. There are several good books on functional programming, including [Hen87] [BW88] [Hen80] [SF89]. For an introduction that also treats implementation issues, see [Pey87]. The articles in [Tur90a] make an interesting introduction to some research topics.

John Backus, one of the designers of Fortran, proposed a new language for functional programming without any names called FP in his Turing Award Lecture [Bac78a]. A functional programming style using a more congenial notation based on Landin’s ISWIM [Lan66] is developed in Henderson’s book *Functional Programming: Application and Implementation* [Hen80].

More recent languages have emphasized lazy evaluation. Haskell and its predecessor Miranda [Tur90a] are good examples. Miranda also has an interesting notion of data abstraction.

4.2 Logic Programming

Kowalski’s paper “Algorithm = Logic + Control” is a good introduction to logic programming in an idealized setting [Kow79]. A classic textbook on Prolog is the book by Clocksin and Mellish [CM81]; another good text is Sterling and Shapiro’s [SS94]. A short description and evaluation of Prolog is found in the paper “The Prolog Phenomenon” [McD80]. A survey is [Apt90].

Several “AI languages” preceded the development of Prolog; for example, Planner [SWC71] and Conniver [MS74].

Much work has focused on concurrent logic programming languages [Sha89], which are perhaps more like CSP than logic programming. An evaluation of the Fifth-Generation project and some history of concurrent logic programming languages is found in [SW93]. A related language is Andorra Prolog [BHW89].

Work on type checking for logic programming languages is surveyed in the collection [Pfe92]. The language cProlog features type inference, type checking, and higher-order programming constructs [NM88] [Mi90] [Mi89] [NM90] [MNPS91].

4.3 Other Declarative Programming Paradigms

Constraint-based languages will probably be important in the future. An early attempt was embodied in Steele and Sussman’s work, as described in [SS80]. An overview is given in Leler’s book [Lel88].

Languages based on term rewriting without logic variables also allow for parallelism. A standard example is OBJ [FGJM85], which also has an interesting module system [Gog84].

4.4 Object-Oriented Programming

A good, but not very technical, introduction to object-oriented concepts is given by Cox [Cox86]; his book also discusses the language Objective-C. A more technical introduction is Budd’s book [Bud91]. Meyer’s book on Eiffel also has more technical meat [Mey88], as well as a focus on software engineering concerns. Another fairly complete treatment is given in Goldberg and Robson’s book on Smalltalk-80 [GR83]. A graduate-level introduction is [BGHS91].

Descriptions of object-oriented design methods are found in [Boo91] [WBWW90] [dCLF92] [dCF92] [dCLF93]. Both [Boo91] and [dCLF93] have many references. A treatment of object-oriented design that focuses more on C++ is found in [Mul89]. See also the survey by [WBJ90]. Much recent work in this area has focused on design patterns [Joh92] [GHJV95].

A collection of papers is found in [Pet87]. More collections of edited research papers are [SW87] [KL89].

The major annual conferences on object-oriented programming are the European Conference on Object-Oriented Programming (ECOOP) and Object-Oriented Programming Systems, Languages and Applications (OOPSLA). The ECOOP tends to be more academic, while OOPSLA is more practical. The OOPSLA proceedings have been published as special issues of ACM *SIGPLAN Notices* (November 1986, December 1987 with an addendum in May 1988, and November 1988, and October of the following years). ECOOP and OOPSLA had a joint conference in 1990.
There are now three journals devoted to object-oriented programming. The Journal of Object-Oriented Programming (JOOP) is the oldest. Two more academic journals are Theory and Practice of Object Systems (TAPOS), and Object-Oriented Systems. The best documented and the cleanest object-oriented programming language is Smalltalk-80 [Ing81] [GR83].

A more widely used language is C++ [ES90]. Good introductions to C++ include [Str91] and [Lip91].

Multiple inheritance, a feature not found in Smalltalk-80, seems to be quite useful. Snyder’s analysis of the design issues involved is insightful [Sny86], although his viewpoint is different from that of most advocates of multiple inheritance.

Semantics of inheritance (as opposed to type theory or semantics of subtyping) appear in [Kam88] [BC90] [CP89].

My own views on the subject of object-oriented specification, verification, and subtyping can be found in [IW90] [Lea91b] [IW95]. For a contrast, see also [BW90b] [Ame87] [AvdL90] [Mey88] [MOM90] [Coo92] [UR92] [IW94].

The concept of delegation is explored in [Lie86], in Actra [LTP86] [LaL89], and in Ungar’s language “Self” [US87].

The contrast between message passing and other kinds of polymorphism is one of binding time. Some relevant semantic models are discussed in [Mit90a] and [CHC90].

A somewhat dated survey of the literature on object-oriented programming is [Lea91a].

5 Language Case Studies

The following entries are intended to be selective rather than comprehensive. Instead they are biased towards the most interesting languages and references for the programming language designer. Thus, although COBOL [ANS74] was (is?) the most widely used language on the planet, its influence on programming language design has been small. While the languages discussed below are often obscure, they demonstrate interesting issues in language design.

Those interested in history for its own sake, or in delving further into early languages, should look at the proceedings of the two History of Programming Languages Conferences [Wex78] [Wex93]. The first conference covers the earliest languages, including COBOL, BASIC, and many others not discussed below. Many of the more established languages have their definitions standardized. These are often published by the American National Standards Institute (ANSI), the International Standards Organization (ISO), or the IEEE.

Other resources for case studies include Kamin’s book [Kam90], which has several case studies put in a common framework. The “Grand Tour” book by Horowitz has articles about specific languages as well [Hor87]. See also the languages mentioned under the various paradigms above.

5.1 FORTRAN

The first widely used programming language was FORTRAN. See [Bac78b] for a discussion of the history of FORTRAN, what early versions of FORTRAN were like, and early references. The development of FORTRAN IV is discussed in [BH64].

John Backus, who headed the team that developed FORTRAN, later became dissatisfied with the influence that FORTRAN had on programming languages [Bac78a].

5.2 Algol 60

The Algol 60 report is a true classic [NBB+63]. Among other innovations, it introduced the syntax formalism now known as BNF. Despite the precise use of English in the report, Knuth and others were able to find problems with the language definition [Knu67].
5.3 Algol 68

Algol 68 is a direct descendent of Algol. It is a more powerful and more complete language than Algol 60; for example, it has user-defined types, overloading of operators, and mechanisms for parallel processing. The language design is fascinating and bristles with examples of orthogonality (one of many terms coined in the Algol 68 design). The revised report is a forbidding document, which has an innovative formal mechanism for defining the language’s semantics [vWMP+77]. Because the revised report is difficult to follow, Tanenbaum’s tutorial is probably a better place to start [Tan76]. Those seriously interested in Algol 68 will want to consult [LvdM77].

5.4 C

A popular descendent of Algol 68 is the lower-level language C [KR78]. C represents the best of several languages that support low-level programming while maintaining the portability of the resulting program. The language has recently been standardized, and incorporates several changes from its variant C++ [Str91].

5.5 Algol W, Pascal, Modula-2, and Oberon

Wirth and Hoare’s language known as Algol W can be thought of as an improved version of Algol 60 [WH66]. Wirth’s language Pascal [Wir71] [JW74] has been enormously popular, attracting detractors [Hab73] and defenders [LD75]. Pascal is, in part, a response to the complexity of Algol 68. All this attention has provoked Wirth to reassessing Pascal [Wir75] and to the design of Modula-2 [Wir85], and Oberon [Wir88]. Oberon has object-oriented features, as does the (non-Wirth) language Modula-3 [CDJ+89] [Nel91].

5.6 Euclid

Euclid is an attempt to improve on Pascal in a different direction [PHL+77] [LGH+78]. Specifically, it attempts to support program verification.

5.7 Ada

Ada was designed by first setting out requirements for the language [Hig78] and then designing and revising a language to meet those requirements [IBH+78] [Ada83] [IBFW91]. Recently, the language was revised to add some object-oriented features [BB+95] [Ada95].

5.8 Lisp-like Languages

The original LISP is described in the *LISP 1.5 Programmer’s Manual* [MAE+65]. The language has since evolved in many directions. MacLisp is a main-stream dialect that provided many system building tools [Pit83]. The successor to MacLisp is Common Lisp [Ste84] [Ste90]. Unlike most earlier dialects of Lisp, Common Lisp has static scoping.

Scheme was the first dialect of Lisp to emphasize static scoping [SS78b] [RCA+86] [ASS85]. See also the references in the *Revised* report [CE91].

ZetaLisp (as found on Symbolics Lisp Machines) includes the influential *Flavors* mechanism for object-oriented programming [WM80] [SMW84] [Sym84]. The Flavors mechanism evolved into the Common Lisp Object System [Kee89] [Ste90] [Pae93]. The meta-object system of Common Lisp is described in [KdRB91]; it provides a very flexible way to extend the language.

5.9 Snobol, SL5, and Icon

SNOBOL4 is a language for string processing that is unlike any other [GPP71]. Because it is so unstructured, its designers have tried to place its powerful features in a more structured framework. The language SL5, one attempt in this direction, is notable for its flexible procedure mechanism [HG78]. Another descendent of SNOBOL4, Icon, is a more conventional programming language with innovative control structures [GG83].
5.10 APL

Another unconventional language is APL [Ive62, Ive91]. APL has array processing features without equal and generic operators that can be combined in interesting ways [Ive79]. More references on APL can be found by consulting the proceedings of the yearly International Conference on APL (sponsored by the ACM). (In the past this was often just called APL 83 or some such name.)

The most recent version of APL, APL2, is described in [Bro88, BC91].

6 Parallel and Distributed Programming Languages

Parallel programming languages are a hot topic of current research, and one with considerable overlap with operating systems, networking, and database systems. Many of the above areas and articles have implications for parallel programming. This area also has several journals and conferences of its own. Some journals that publish programming language related materials but which were not mentioned above include Distributed Computing, IEEE Transactions on Parallel and Distributed Systems, and International Journal of Parallel Programming. See also ACM SIGOPS Operating Systems Review. Important conferences not mentioned above include the annual ACM Symposium on Operating Systems Principles, ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming, and the ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing.

An older, but still good, survey on parallel programming language issues is [AS83]. A survey that focuses on the Linda model appears in [CG91]. (See [BZ91, BZ92, GCL92] for more work on Linda.) A survey that focuses on object-oriented aspects is [WKH92]. A survey that focuses on distributed programs is [And91b]. Andrew’s textbook on concurrent programming languages also contains material on program verification as well as some survey material [And91a]. Data flow languages are surveyed in [Ack82]. A useful collection of papers is [ST95]. An introduction to concurrent programming techniques is [Sno92].

Actor languages are discussed in [Agh91]. A concurrent version of ML is described in [Rep93]. Some aspects of data parallel programming languages are described in [HS86, Gri93]. The language Jade is described in [RL92, RSL93].

Distributed programming is a subarea of parallel programming with its own set of problems [Mul93]. A survey appears in [BST89], which can be consulted for other references. A survey, with a focus on object-orientation is [CC91]. Other sources of references are the collection of reprints [AS91], and the book [GR93]. Of particular interest are the languages Argus [LS83, Wei90], SR [AOC+88, AO93], and Obliq [Car95]. The Amoeba distributed operating system also has interesting implications for language designers [TvrS+90, TKB92].

An older but still heavily used approach to the semantics of concurrent processes is Petri nets [Pet77, PC92]. A classic reference for the operational semantics of concurrent processes is Milner’s book on CCS [Mil80]. Another widely used approach is Hoare’s CSP [Hoa78, BHR84, LS84, Hon85]. See [Hen88, BM90, Mil90] for more work in the semantics of concurrency, and [LL90] and [Bro91] for work in the semantics of distributed systems.

7 The Future

A (by now a bit dated) summary of research directions for language design is given in [LLM89].

Consumers of programming languages (programmers and language standardization committees) seem to be fairly conservative, and interested more in performance than elegance or expressive power. See [Gab93] for a pessimistic view of what this means for language design.
References


25


