Final Program

The Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III)

San Diego, California, USA
9-10 June 2007

in cooperation with ACM SIGSOFT

(co-located with FCRC 2007, 9-16 June 2007)
Table of Contents

Co-chairs Introduction .............................................................. 5
Conference Agenda ........................................................................ 9

AppleScript .................................................................................... 12
The evolution of Lua ....................................................................... 14
A history of Modula-2 and Oberon .............................................. 16
Evolving a language in and for the real world:
  C++ 1991-2006 ......................................................................... 18
Statecharts in the making: a personal account ......................... 20
A history of Erlang ......................................................................... 22
The rise and fall of High Performance Fortran:
  an historical object lesson ......................................................... 24
The design and development of ZPL ......................................... 26
Self ................................................................................................. 28
The when, why and why not of the BETA
  programming language ............................................................. 30
The development of the Emerald programming language .......... 32
A history of Haskell: being lazy with class ............................... 34

Appendix: ....................................................................................... 37
History of Programming Languages Conference: HOPL-III  
Co-Chairs Introduction

In 1978, at the first HOPL conference, Jean Sammet wrote:

I'm happy to have this opportunity to open this Conference on the History of Programming Languages. It's something that I personally have been thinking about for almost seven years although the actual work on this has only been going on for the past one-and-a-half years. Most of you have not had the opportunity to look at the Preprints, and perhaps you haven't had too much time to even look at the program. For that reason I want to make sure that you understand something about what is intended, and frankly what is not intended be done in this conference. We view this as a start: perhaps the first of a number of conferences which will be held about various aspects of the computing field in general, software in particular, and even more specifically, programming languages. We hope that this conference and the Preprints and the final proceedings will stimulate many older people, many younger people, and many of those in the in-between category in the field to do work in history.

This conference, I repeat again, is certainly not the last set of words to be held on this subject. It's only a beginning. I want to emphasize also that it's not a conference on the entire history of programming languages, nor even on the entire history of the languages that we selected. As many of you have seen from some of the earlier publicity that we put out, we're trying to consider and report on the technical factors which influenced the development of languages which satisfied a number of criteria. First, they were created and in use by 1967; they remain in use in 1977, which is when we made the decisions; and they've had considerable influence on the field of computing. The criteria for choosing a language include the following factors, although not every factor applied to each language: we considered usage, influence on language design, overall impact on the environment, novelty, and uniqueness. Particularly because of the cut-off date of 1967, some languages, which are in common use today, are not included. We definitely wanted a perspective of 10 years before we started worrying about the early history of the language.

HOPL-I was a start and it did stimulate (some older and younger) people to continue the work of documenting the history of computing in general, and programming languages, in particular. HOPL-II followed in 1993. It extended the notion of history from HOPL-I to include the evolution of languages, language paradigms, and language constructs. It preserved the 10-year perspective. It also repeated the HOPL-I multi-year preparation of submissions, reviews, and re-reviews with teams of reviewers and experts, to come up with the best possible history papers from the people who were directly involved with the creation of their languages.
Fifteen years later, HOPL-III is another step in the documentation of the history of our field. Work began three years ago in 2004 to create a Program Committee, to establish paper solicitation criteria (see appendix to this proceedings), and to encourage submissions. As with its predecessors, the goal of HOPL-III was to produce an accurate historical record of programming language design and development. To achieve this goal, the Program Committee worked closely with prospective authors and outside experts to help ensure that all the papers were of high quality. As with HOPL-I and II, there were multiple rounds of reviewing to ensure that all the selected papers met requirements for both technical accuracy and historical completeness.

The criteria for the programming languages considered appropriate for HOPL-III were:

1. The programming language came into existence before 1996, that is, it was designed and described at least 11 years before HOPL-III (2007).

2. The programming language has been widely used since 1998 either (i) commercially or (ii) within a specific domain. In either case, “widely used” implies use beyond its creators.

3. There also are some research languages which had great influence on widely used languages that followed them. As long as the research language was used by more than its own inventors, these will be considered to be appropriate for discussion at HOPL-III.

The twelve papers in this proceedings represent original historical perspectives on programming languages that span at least five different programming paradigms and communities: object-oriented, functional, reactive, parallel, and scripting. At the time of the conference, the programming languages community continues to create broader mini-histories of each of those paradigms at http://en.wikipedia.org/wiki/HOPL

A conference of this scope and level of preparation could not have happened without the time and assistance of many, many people. First we must thank our colleagues on the program committee

Fran Allen, IBM Research (Emerita)
Thomas J. (Tim) Bergin, American University (Emeritus)
Andrew Black, Portland State University
Koen Claessen, Chalmers University of Technology
Kathleen Fisher, AT&T Research
Susan L. Graham, University of California, Berkeley
Final Program

Julia Lawall, DIKU
Doug Lea, SUNY Oswego
Peter Lee, Carnegie Mellon University
Michael S. Mahoney, Princeton University
Guy Steele, Sun Microsystems
Benjamin Zorn, Microsoft Research

and the authors of all the submitted papers.

We must also thank the language experts who helped with the extensive paper reviews: Chris Espinoza, Gilad Bracha, Herb Sutter, Andrew Watson, Ulf Wiger, Vivek Sarkar, Norman Ramsey, Greg Nelson, Craig Chambers, and Kathy Yellick. We also thank the set of experts who helped seed the Wikipedia discussions of the language paradigms: Vijay Saraswat, Bard Bloom, Dipayan Gangopadhyay, and Guido van Rossum. Finally, we would like to thank the staff at ACM Headquarters, the SIGPLAN Executive Committee, the SIGSOFT Executive Committee, and Diana Priore, all of whom made this complex conference possible, and Joshua Hailpern, who designed both the HOPL CACM advertisement and the Proceedings and Final Program cover art.

We also wish to acknowledge the generous financial support for HOPL-III that has been provided by:

- An anonymous donor for multimedia capture/post-processing
- Microsoft Research for manuscript copy-editing and proceedings preparation
- IBM Research for subsidizing student registration and in support of program committee operations, the final program, and the HOPL-III website
- ACM SIGPLAN Executive Committee

Microsoft Research  SIGPLAN  IBM Research

This has been an ambitious and lengthy project for us; we are glad to see it successfully completed. We hope you enjoy both the conference presentations and the papers in these proceedings – a (partial) record of the past 15 years of our programming languages community.

Barbara Ryder, Rutgers University
Brent Hailpern, IBM Research
HOPL-III Conference/Program Committee co-Chairs
Final Program
HOPL-III Agenda: Saturday, 9 June 2007

08:45 - 09:00  Introduction

09:00 - 10:00  Keynote by Guy Steele (Sun Microsystems) and Richard P. Gabriel (IBM Research)

10:00 - 10:30  Break

10:30 - 12:20  "AppleScript" by William R. Cook (University of Texas at Austin)

   "The evolution of Lua" by Roberto Ierusalimschy (PUC-Rio), Luiz Henrique de Figueiredo (IMPA), and Waldemar Celes (PUC-Rio)

12:20 - 13:30  Lunch

13:30 - 15:20  "A history of Modula-2 and Oberon" by Niklaus Wirth (ETH Zurich)

   "Evolving a language in and for the real world: C++ 1991–2006" by Bjarne Stroustrup (Texas A&M University and AT&T Labs - Research)

15:20 - 16:00  Break

16:00 - 17:50  "Statecharts in the making: a personal account" by David Harel (Weizmann Institute of Science)

   "A history of Erlang" by Joe Armstrong (Ericsson AB)
HOPL-III Agenda: Sunday, 10 June 2007

08:00 - 08:15  Introduction

08:15 - 10:05  "The rise and fall of High Performance Fortran: an historical object lesson" by Ken Kennedy (Rice University), Charles Koelbel (Rice University), Hans Zima (Institute of Scientific Computing, University of Vienna and Jet Propulsion Laboratory, California Institute of Technology)

"The design and development of ZPL" by Lawrence Snyder (University of Washington)

10:05 - 10:30  Break

10:30 - 12:20  "Self" by David Ungar (IBM Research), Randall B. Smith (Sun Microsystems)

"The when, why and why not of the BETA programming language" by Bent Bruun Kristensen (University of Southern Denmark), Ole Lehrmann Madsen (University of Aarhus), Birger Møller-Pedersen (University of Oslo)

12:20 - 13:30  Lunch

13:30 - 15:20  "The development of the Emerald programming language" by Andrew P. Black (Portland State University), Norman C. Hutchinson (University of British Columbia), Eric Jul (University of Copenhagen) and Henry M. Levy (University of Washington)

"A History of Haskell: being lazy with class" by Paul Hudak (Yale University), John Hughes (Chalmers University), Simon Peyton Jones (Microsoft Research), and Philip Wadler (University of Edinburgh)

15:20 - 15:40  Break

15:40 - 17:00  Panel: Programming Language Paradigms: Past, Present, and Future

  Kathleen Fisher (AT&T Labs - Research), chair
  Bertrand Meyer (ETH Zurich)
  Olin Shivers (Georgia Institute of Technology)
  Larry Wall
  Kathy Yelick (University of California, Berkeley)
AppleScript is a scripting language and environment for the Mac OS. Originally conceived in 1989, AppleScript allows end users to automate complex tasks and customize Mac OS applications. To automate tasks, AppleScript provides standard programming language features (control flow, variables, data structures) and sends Apple Events to invoke application behavior. Apple Events are a variation on standard remote procedure calls in which messages can identify their arguments by queries that are interpreted by the remote application. This approach avoids the need for remote object pointers or proxies, and reduces the number of communication round trips, which are expensive in high latency environments like the early Macintosh OS. To customize an application that uses AppleScript's Open Scripting Architecture, users attach scripts to application objects; these scripts can then intercept and modify application behavior.

AppleScript was designed for casual users: AppleScript syntax resembles natural language, and scripts can be created easily by recording manual operations on a graphical interface. AppleScript also supported internationalization in allowing script to be presented in multiple dialects, including English, Japanese, or French. Although the naturalistic syntax is easy to read, it can make scripts much more difficult to write.

Early adoption was hindered by the difficulty of modifying applications to support Apple Events and the Open Scripting Architecture. Yet AppleScript is now widely used and is an essential differentiator of the Mac OS. AppleScript's communication model is a precursor to web services, and the idea of embedded scripting has been widely adopted.
The Evolution of Lua

Roberto Ierusalimschy\textsuperscript{1}, Luiz Henrique de Figueiredo\textsuperscript{2}, and Waldemar Celes\textsuperscript{1}

\textsuperscript{1}Department of Computer Science  
PUC-Rio, Rio de Janeiro, Brazil  
roberto@inf.puc-rio.br  
celes@inf.puc-rio.br

\textsuperscript{2}IMPA—Instituto Nacional de  
Matem\'atica Pura e Aplicada, Brazil  
lhf@impa.br

Lua is a scripting language born in 1993 at PUC-Rio in Brazil. Since its origins as an in-house configuration language, Lua has evolved far beyond our most optimistic expectations to become widely used in all kinds of industrial applications. In particular, Lua is one of the leading languages in game scripting.

From the start, Lua was designed to be simple, small, portable, fast, and easily embedded into applications. We believe that these design principles account for Lua’s success in industry. The main characteristic of Lua is that it offers a single kind of data structure, the table, which is the Lua term for an associative array. Although most scripting languages offer associative arrays, in no other language do associative arrays play such a central role. Lua tables provide simple and efficient implementations for modules, classes, objects, and most common data structures.

We shall report on the birth and evolution of Lua, and discuss how Lua moved from a simple configuration language to a versatile, widely used language that supports extensible semantics, anonymous functions, full lexical scoping, proper tail calls, and coroutines.
This is an account of the development of the languages Modula-2 and Oberon. Together with their ancestors ALGOL 60 and Pascal they form a family called Algol-like languages. Pascal (1970) reflected the ideas of structured programming, Modula-2 (1979) added those of modular system design, and Oberon (1988) catered to the object-oriented style. Thus they mirror the essential programming paradigms of the past decades. Here the major language properties are outlined, followed by an account of the respective implementation efforts. The conditions and the environments in which the languages were created are elucidated. We point out that simplicity of design was the most essential guiding principle. Clarity of concepts, economy of features, efficiency and reliability of implementations were its consequences.
Evolving a language in and for the real world: 
C++ 1991-2006

Bjarne Stroustrup
Texas A&M University
www.research.att.com/~bs

This paper outlines the history of the C++ programming language from the early
days of its ISO standardization (1991), through the 1998 ISO standard, to the
later stages of the C++0x revision of that standard (2006). The emphasis is on
the ideals, constraints, programming techniques, and people that shaped the
language, rather than the minutiae of language features. Among the major
themes are the emergence of generic programming and the STL (the C++
standard library is algorithms and containers). Specific topics include separate
compilation of templates, exception handling, and support for embedded systems
programming. During most of the period covered here, C++ was a mature
language with millions of users. Consequently, this paper discusses various uses
of C++ and the technical and commercial pressures that provided the
background for its continuing evolution.
Statecharts in the Making: A Personal Account

David Harel
The Weizmann Institute of Science
Rehovot, ISRAEL 76100
dharel@weizmann.ac.il

This paper is a highly personal and subjective account of how the language of statecharts came into being. The main novelty of the language is in being a fully executable visual formalism intended for capturing the behavior of complex real-world systems, and an interesting aspect of its history is that it illustrates the advantages of theoreticians venturing out into the trenches of the real world, "dirtying their hands" and working closely with the system's engineers. The story is told in a way that puts statecharts into perspective and discusses the role of the language in the emergence of broader concepts, such as visual formalisms in general, reactive systems, model-driven development, model executability and code generation.
A History of Erlang

Joe Armstrong
Ericsson AB
joe.armstrong@ericsson.com

Erlang was designed for writing concurrent programs that "run forever." Erlang uses concurrent processes to structure the program. These processes have no shared memory and communicate by asynchronous message passing. Erlang processes are lightweight and belong to the language, not the operating system. Erlang has mechanisms to allow programs to change code "on the fly" so that programs can evolve and change as they run. These mechanisms simplify the construction of software for implementing non-stop systems.

This paper describes the history of Erlang. Material for the paper comes from a number of different sources. These include personal recollections, discussions with colleagues, old newspaper articles and scanned copies of Erlang manuals, photos and computer listings and articles posted to Usenet mailing lists.
The Rise and Fall of High Performance Fortran: 
An Historical Object Lesson

Ken Kennedy¹, Charles Koelbel¹, and Hans Zima²

¹Rice University, Houston, TX
chk@rice.edu

²Institute of Scientific Computing,
University of Vienna, Austria,
and
Jet Propulsion Laboratory,
California Institute of Technology,
Pasadena, CA
zima@jpl.nasa.gov

High Performance Fortran (HPF) is a high-level data-parallel programming system based on Fortran. The effort to standardize HPF began in 1991, at the Supercomputing Conference in Albuquerque, where a group of industry leaders asked Ken Kennedy to lead an effort to produce a common programming language for the emerging class of distributed memory parallel computers. The proposed language would focus on data-parallel operations in a single thread of control, a strategy which was pioneered by some earlier commercial and research systems, including Thinking Machines’ CM Fortran, Fortran D, and Vienna Fortran.

The standardization group, called the High Performance Fortran Forum (HPFF), took a little over a year to produce a language definition that was published in January 1993 as a Rice technical report [50] and, later that same year, as an article in Scientific Programming [49].

The HPF project had created a great deal of excitement while it was underway and the release was initially well received in the community. However, over a period of several years, enthusiasm for the language waned in the United States, although it has continued to be used in Japan.

This paper traces the origins of HPF through the programming languages on which it was based, leading up to the standardization effort. It reviews the motivation underlying technical decisions that led to the set of features incorporated into the original language and its two follow-ons: HPF 2 (extensions defined by a new series of HPFF meetings) and HPF/JA (the dialect that was used by Japanese manufacturers and runs on the Earth Simulator).

A unique feature of this paper is its discussion and analysis of the technical and sociological mistakes made by both the language designers and the user community; mistakes that led to the premature abandonment of the very promising approach employed in HPF. It concludes with some lessons for the future and an exploration of the influence of ideas from HPF on new languages emerging from the High Productivity Computing Systems program sponsored by DARPA.
The Design and Development of ZPL

Lawrence Snyder
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195-2350
snyder@cs.washington.edu

ZPL is an implicitly parallel programming language, which means all instructions to implement and manage the parallelism are inserted by the compiler. It is the first implicitly parallel language to achieve performance portability, that is, consistent high performance across all (MIMD) parallel platforms. ZPL has been designed from first principles, and is founded on the CTA abstract parallel machine. A key enabler of ZPL’s performance portability is its What You See Is What You Get (WYSIWYG) performance model. The paper describes the antecedent research on which ZPL was founded, the design principles used to build it incrementally, and the technical basis for its performance portability. Comparisons with other parallel programming approaches are included.
The years 1985 through 1995 saw the birth and development of the language Self, starting from its design by the authors at Xerox PARC, through first implementations by Ungar and his graduate students at Stanford University, and then with a larger team formed when the authors joined Sun Microsystems Laboratories in 1991. Self was designed to help programmers become more productive and creative by giving them a simple, pure, and powerful language, an implementation that combined ease of use with high performance, a user interface that off-loaded cognitive burden, and a programming environment that captured the malleability of a physical world of live objects. Accomplishing these goals required innovation in several areas: a simple yet powerful prototype-based object model for mainstream programming, many compilation techniques including customization, splitting, type prediction, polymorphic inline caches, adaptive optimization, and dynamic deoptimization, the application of cartoon animation to enhance the legibility of a dynamic graphical interface, an object-centered programming environment, and a user-interface construction framework that embodied a uniform use-mention distinction. Over the years, the project has published many papers and released four major versions of Self.

Although the Self project ended in 1995, its implementation, animation, user interface toolkit architecture, and even its prototype object model impact computer science today (2006). Java virtual machines for desktop and laptop computers have adopted Self’s implementation techniques, many user interfaces incorporate cartoon animation, several popular systems have adopted similar interface frameworks, and the prototype object model can be found in some of today’s languages, including JavaScript. Nevertheless, the vision we tried to capture in the unified whole has yet to be achieved.
The When, Why and Why Not of the BETA Programming Language

This paper tells the story of the development of BETA: a programming language with just one abstraction mechanism, instead of one abstraction mechanism for each kind of program element (classes, types, procedures, functions, etc.). The paper explains how this single abstraction mechanism, the pattern, came about and how it was designed to be so powerful that it covered the other mechanisms.

In addition to describing the technical challenge of capturing all programming elements with just one abstraction mechanism, the paper also explains how the language was based upon a modeling approach, so that it could be used for analysis, design and implementation. It also illustrates how this modeling approach guided and settled the design of specific language concepts.

The paper compares the BETA programming language with other languages and explains how such a minimal language can still support modeling, even though it does not have some of the language mechanisms found in other object-orientated languages.

Finally, the paper tries to convey the organization, working conditions and social life around the BETA project, which turned out to be a lifelong activity for Kristen Nygaard, the authors of this paper, and many others.
The Development of the Emerald Programming Language

Emerald was a research programming language developed at the University of Washington from 1983 to 1987. It was originally implemented on DEC VAX hardware, later on the Motorola 68000 and SPARC, and then on a portable virtual machine. Emerald was object-based, in the sense that it provided objects but not classes or inheritance; its distinguishing feature was support for distribution, including mobile objects and processes. Along the way, Emerald also made significant strides in type theory, including implementations of what have since become known as F-bounded polymorphism and generic data structures.

Emerald achieved all of this, and excellent performance, by decoupling the way that an object was defined in the programming language from the way that it was implemented. Contemporary object-based distributed languages had two notions of object: “small objects”, which were efficiently supported within a single address space, but could not be accessed remotely, and “large objects”, which were accessible from remote address spaces but which were much more costly. One of Emerald’s contributions was the realization that these different implementations should not be visible in the source language, which needed only a single notion of object, even though they were essential in the generated code.

Emerald’s type system thoroughly separated an object’s implementation from its type. Types characterized the operations that an object could understand, along with their argument and result types; many different implementations could conform to the same type.

Letting the compiler choose an implementation was only possible if this did not change the semantics, so we were forced to define the semantics abstractly. Thus, the semantics of parameter passing could not depend on the relative locations of the invoking and invoked objects. Emerald did include features to control the location of objects: to fix and unfix an object at a specific location, to move an object to a remote location with an invocation, and to cause an object to “visit” a remote location for the duration of an invocation. Using these features did not change the semantics of a program, but might dramatically change its performance. Emerald also distinguished functions (which had no effect on the state) from more general operations, and immutable objects (which the implementation was free to replicate) from mutable objects (which at any one time existed on a single host).

Although Emerald’s support for remote invocation has been widely reproduced, remote invocation has rarely been implemented with such semantic transparency. Implementations of mobile objects are still rare, and languages that incorporate mobility into their semantic model rarer still.
A History of Haskell: Being Lazy With Class

Paul Hudak
Yale University
paul.hudak@yale.edu

John Hughes
Chalmers University
rjmh@cs.chalmers.se

Simon Peyton Jones
Microsoft Research
simonpj@microsoft.com

Philip Wadler
University of Edinburgh
wadler@inf.ed.ac.uk

Functional programming, particularly in its purely functional form, is a radical and principled attack on the challenge of writing programs that work. Great progress has been made in improving the expressiveness of such languages, and in making them practical and suitable for a wide spectrum of applications. The non-strict purely functional language Haskell, conceived almost twenty years ago, has contributed to that progress by sticking remorselessly to the discipline of purity, and by building a critical mass of interest and research effort behind a single language.

This paper describes the history of Haskell in four parts. First Haskell’s genesis and principles are discussed, including themotivation for designing a new functional language, the formation of an international committee to do so, anecdotes surrounding the first meetings, and Haskell's relationship to Miranda. In part two the technical contributions are summarized, including an historical account of the development of Haskell's key innovative features: type classes and monads. Implementations and tools are discussed in part three, beginning with early implementations and ending with the most widely used today, and including a technical discussion of profiling and debugging. In the last part of the paper, applications of Haskell are summarized and an assessment is made of its impact on education, the real world, and on other language design efforts.
Appendix: Content Guidelines for Authors

The criteria for the programming languages considered appropriate for HOPL-III are:

1. The programming language came into existence before 1996, that is, it was designed and described at least 11 years before HOPL-III (2007).

2. The programming language has been widely used since 1998 either (i) commercially or (ii) within a specific domain. In either case, “widely used” implies use beyond its creators.

3. There also are some research languages which had great influence on widely used languages that followed them. As long as the research language was used by more than its own inventors, these will be considered to be appropriate for discussion at HOPL-III.

Please be sure to include within your paper a clear indication of how the subject material satisfies these criteria (i.e., 1&2 or 1&3). This information can certainly be provided indirectly as part of the overall text. (For instance, some of the criteria are dates; it suffices to include the dates as part of the historical narrative.)

The main purpose of these guidelines is to help you develop the appropriate content for your contribution to HOPL-III. The questions herein point to the kind of information that people want to know about the history of programming languages. A set of questions is included for each of the major areas to be covered by HOPL-III:

- Early history of a specific language
- Later evolution of a specific language (usually a language treated in HOPL-I or in HOPL-II)

The sets of questions overlap to some extent. This guideline includes both sets of questions, in the hope that having the other set available may prove useful to you.

Even within a single set, the same question, or very similar questions, may be asked in different contexts. Please draft your paper in light of these different emphases and contexts.

Your paper should try to answer as many questions as possible in your topic area: it is understood that you might not be able to address every one of them. Because history can unfold in so many different ways, some of the questions may be clearly irrelevant to your particular topic, or your particular point of view. The information requested might no longer be available, but please remember this information is important as well. Several questions are of the form “How did something affect something else?”—it can be important for the historical record to assert that “it didn’t.”

The question set suggests the content, not the form, of your paper. (In particular, your paper should not be in question-answer format.) The questions are organized into topics and subtopics for your convenience during your research; this structure is not meant as
an outline for your paper. (Topics, subtopics, and questions are also numbered and lettered for convenience of reference.) Please feel free to use whatever form and style seems most appropriate and comfortable to you.

The Program Committee strongly suggests that you examine at least one paper from each of the proceedings of HOPL-I and HOPL-II to see what earlier contributors did. The references are:

History of Programming Languages, 
Edited by, Richard L. Wexelblat, 
Academic Press, 1981

History of Programming Languages-II, 
Edited by, Thomas J. Bergin, Jr. and Richard G. Gibson, Jr., 

The first two History of Programming Languages conferences established high technical and editorial standards. We trust that your contribution will help HOPL-III maintain or surpass these standards.
QUESTIONS ON THE EARLY HISTORY OF A LANGUAGE

I. BACKGROUND

1. Basic Facts about Project Organization and People

(a) Under what organizational auspices was the language developed (e.g., name of company, department/division in the company, university?) Be as specific as possible about organizational subunits involved.

(b) Were there problems or conflicts within the organization in getting the project started? If so, please indicate what these were and how they were resolved.

(c) What was the source of funding (e.g., research grant, company R&D, company production units, or a government contract?)

(d) Who were the people on the project and what was their relationship to the author(s) (e.g., team members, subordinates)? To the largest extent possible, name all the people involved, including part-timers, when each person joined the project and what each person worked on. Indicate the technical experience and background of each participant, including formal education.

(e) Was the development done originally as a research project, as a development project, as a committee effort, as an open-source effort, as a one-person effort with some minor assistance, or…?

(f) Was there a defined leader to the group? If so, what was his or her exact position (and title) and how did he or she get to be the leader? (e.g., appointed “from above”, self-starter, volunteer, elected?)

(g) Was there a de facto leader different from the defined leader? If so, who was this leader and what made them the de facto leader, i.e., personality, background, experience, a “higher authority” or something else?

(h) Were there consultants from outside the project who had a formal connection to it? If so, who were they, how and why were they chosen, and how much help were they? Were there also informal consultants? If so, please answer the same questions.

(i) Did the participants of the project view themselves primarily as language designers, as implementers, or as eventual users? If there were some of each working on the project, indicate the split as much as possible. How did this internal view of the people involved affect the initial planning and organization of the project?

(j) Did the language designers know (or believe) that they would also have the responsibility for implementing the first version? Whether the answer is “yes” or “no” and was the technical language design affected by this?

2. Costs and Schedules

(a) Was there a budget? Did the budget provide a fixed upper limit on the costs? If so, how much money was to be allocated and in what ways? What external or internal factors led to the budget constraints? Was the money formally divided between language design and actual implementation? If so, indicate in what way?

(b) Was there a fixed deadline for completion of the project? Was the project divided into phases and did these have deadlines? How well were the deadlines met?

(c) What is the best estimate for the amount of human resources involved (i.e., in person-years)? How much was for language design, for documentation, and for implementation?

(d) What is the best estimate of the costs prior to putting the first system in the hands of the first users? If possible, show as much breakdown on this as possible.
(e) If there were cost and/or schedule constraints, how did that affect the language design and in what ways?

3. Basic Facts About Documentation

(a) In the planning stage, was there consideration of the need for documentation of the work as it progressed? If so, was it for internal communication among project members or external monitoring of the project by others, or both?

(b) What types of documentation were decided upon?

(c) To the largest extent possible, cite both dates and documents for the following (including internal papers and web sites which may not have been released outside of the project) by title, date, and author. (In items c1, c4, c9, and c10, indicate the level of formality of the specifications – e.g., English, formal notation – and what kind.)

* (c1) Initial idea
* (c2) First documentation of initial idea
* (c3) Preliminary specifications
* (c4) “Final” specifications (i.e., those which were intended to be implemented)
* (c5) “Prototype” running (i.e., as thoroughly debugged as the state of the art permitted, but perhaps not all of the features included)
* (c6) “Full” language compiler (or interpreter) was running
* (c7) Usage on real problems done by the developers
* (c8) Usage on real problems done by people other than the developers
* (c9) Documentation by formal methods
* (c10) Paper(s) in professional journals or conference proceedings
* (c11) Please identify extensions, modifications and new versions

4. Languages and Systems Known at the Time

(a) What specific languages were known to you and/or other members of the development group at the time the work started? Which others did any of you learn about as the work progressed? How much did you know about these languages and in what ways (e.g., as users, from having read unpublished and/or published papers, informal conversations)? (Please try to distinguish between what you, as the writer knew and what the other members of the project knew.)

(b) Were these languages considered as formal inputs that you were definitely supposed to consider in your own language development, or did they merely provide background? What was it about these languages that you wanted to emulate (e.g., syntax, capabilities, internal structure, application area, etc.)?

(c) How influenced were you by these languages? Put another way, how much did the prior language backgrounds of you and other members of the group influence the early language design? Whether the answer is “a lot” or “a little,” why did these other languages have that level of influence? (This point may be more easily considered in Section II: Rationale of Content of Language.)

(d) Was there a primary source of inspiration for the language and if so, what was it? Was the language modeled after this (or any other predecessors or prototypes)?

5. Intended Purposes and Users

(a) For what application area was the language designed, i.e., what type of problems was it suppose to be used for? Be as specific as possible in describing the application area; for example, was “business data processing” or “scientific applications” ever carefully defined? Was the apparent application area of some other language used as a model?

(b) For what types of users was the language intended (e.g., experienced programmers, mathematicians, business people, novice programmers, non-programmers)? Was there any conflict within the group on this? Were compromises made, and if so, were they made for technical or non-technical reasons?
(c) What equipment was the language intended to be implemented on? Wherever possible, cite specific machine(s) by manufacturer(s) and number, or alternatively, give the broad descriptions of the time period with examples (e.g., “COBOL was defined to be used on ‘large’ machines which at that time included UNIVAC I and II, IBM 705.”) Was machine independence a significant design goal, albeit within this class of machines? (See also Question (1b) in Rationale of the Content of the Language.)

6. Source of Motivation

(a) What (or who) was the real origin of the idea to develop this language?

(b) What was the primary motivation in developing the language (e.g., research, task assigned by management?)

II. RATIONALE OF THE CONTENT OF THE LANGUAGE

These questions are intended to stimulate thought about various factors that affect most language design effort. Not all the questions are relevant for every language. They are intended to suggest areas that might be addressed in each paper.

1. Environment Factors

To what extent was the design of the language influenced by:

(a) Program size: Was it explicitly thought that programs written in the language would be large and/or written by more than one programmer? What features were explicitly included (or excluded) for this reason? If this factor wasn’t considered, did it turn out to be a mistake? Were specific tools or development environments designed at the same time to support these choices?

(b) Program libraries: Were program libraries envisioned as necessary or desirable, and if so, how much provision was made for them?

(c) Portability: How important was the goal of machine independence? What features reflect concern for portability? How well was this goal attained? See also question (1) on Standardization and question (5c) under Background.

(d) User Background and Training: What features catered to the expected background of intended users? In retrospect, what features of the language proved to be difficult for programmers to use correctly? Did some features fall into disuse? Please identify such features and explain why they fell into disuse? How difficult did it prove to train users in the correct and effective use of the language, and was the difficulty a surprise? What changes in the language would have alleviated training problems? Were any proposed features rejected because it was felt users would not be able to use them correctly or appropriately?

(e) Execution Efficiency: How did requirements for executable code size and speed affect the language design? Were programs in the language expected to execute on large or small computers (i.e., was the size of object programs expected to pose a problem)? What design decisions were explicitly motivated by the concern (or lack of concern) for execution efficiency? Did these concerns turn out to be accurate? How was the design of specific features changed to make it easier to optimize executable code?

(f) Target Computer Architecture: To what extent were features in the language dictated by the anticipated target computer, e.g., its word size, existence of floating-point hardware, instruction set peculiarities, availability and use of index registers, special-purpose co-processors and accelerators, etc.?

(g) Compilation Environment: To what extent, if any, did concerns about compilation efficiency affect the design? Were features rejected or included primarily to make it easier to implement compilers for the language or to ensure that the compiler(s) would execute quickly? In retrospect, how correct or incorrect do you feel these decisions were? What decisions did you make regarding use of the compiler run-time system?
(h) Programming Ease: To what extent was the ease of coding an important consideration and what features in the language reflect the relative importance of this goal? Did maintainability considerations affect any design decisions? If so, which ones?

(i) Execution Environment: To what extent did the language design reflect its anticipated use in a batch, embedded, portable, office, or networked environment? What features reflect these concerns?

(j) Parallel Implementation: Were there multiple implementations being developed at the same time as the later part of the language development? If so, was the language design hampered or influenced by this in any way?

(k) Standardization: In addition to (or possibly separate from) the issue of portability, what considerations were given to possible standardization? What types of standardization were considered, and what groups were involved and when?

(l) Networking/Parallel Environment: To what extent did the language design reflect its anticipated use in a networked- or parallel-execution environment? What features reflect these concerns?

2. Functions to be Programmed

(a) How did the operations and data types in the language support the writing of particular kinds of algorithms?

(b) What features might have been left out, if a slightly different application area has been in mind?

(c) What features were considered essential to properly express the kinds of programs to be written?

(d) What misconceptions about application requirements turned up that necessitated redesign of these application specific features before the language was actually released?

3. Language Design Principles

(a) What consideration, if any, was given to designing the language so that programming errors could be detected early and easily? Were the problems of debugging and testing considered? Were debugging and testing facilities deliberately included in the language?

(b) To what extent was the goal of keeping the language simple considered important? What kind of simplicity was considered most important? What did your group mean by “simplicity”?

(c) What thought was given to make programs more understandable and how did these considerations influence the design? Was there conscious consideration of making programs “easy to read” versus “easy to write”? If so, which were chosen and why?

(d) Did you consciously develop the data types first and then the operations, or did you use the opposite order, or did you try to develop both in parallel with appropriate iteration? Were data and operations combined into objects?

(e) To what extent did the design reflect a conscious philosophy of how languages should be designed (or how programs should be developed)? What was this philosophy?

4. Language Definition

(a) What techniques for defining languages were known to you? Did you use these or modify them, or did you develop new ones?

(b) To what extent— if any— was the language itself influenced by the technique used for the definition?
Final Program

5. Concepts About Other Languages

(a) Were you consciously trying to introduce new concepts? If so, what were they? Do you feel that you succeeded?

(b) If you were not trying to introduce new concepts, what was the justification for introducing this new language? (Such justification might involve technical, political, or economic factors.)

(c) To what extent did the design consciously borrow from previous language designs or attempt to correct perceived mistakes in other languages?

6. Influence of Non-technical Factors

(a) How did time and cost constraints (as described in the Background section) influence the technical design?

(b) How did the size and structure of the design group affect the technical design?

(c) Provide any other information you have pertaining to ways in which the technical language design was influenced or affected by non-technical factors.

III. A POSTERIORI EVALUATION

1. Meeting of Objectives

(a) How well do you think the language met its original objectives?

(b) Do the users think the language has met its objectives?

(c) How well do you think the computing community (as a whole) thinks the objectives were met?

(d) How much impact did portability (i.e., machine independence) have on the acceptance by users?

(e) Did the objectives change over time? If so, how, when, and in what ways did they change? See also question (2d) under Rationale of Content of Language and answer here if appropriate.

2. Contributions of Language

(a) What is the major contribution made by this language? Was this one of the objectives? Was this contribution a technical or a non-technical contribution, or both? What other important contributions are made by this language? Were these part of the define objectives? Were these contributions technical or non-technical?

(b) What do you consider the best points of the language, even if they are not considered to be a contribution to the field (i.e., what are you proudest of, regardless of what anybody else thinks)?

(c) How many other people or groups decided to implement this language because of its inherent value?

(d) Did this language have any effect on the development of later hardware?

(e) Did this language spawn any “dialects”? If so, please identify them. Were they major or minor changes to the language definition? How significant did the dialects themselves become?

(f) In what way do you feel the computer field is better off (or worse) for having this language?

(g) What fundamental effects on the future of language design resulted from this language development (e.g., theoretical discoveries, new data types, new control structures)?
3. Mistakes or Desired Changes

(a) What mistakes do you think were made in the design of the language? Were any of these able to be corrected in a later version of the language? If you feel several mistakes were made, list as many as possible with some indication of the severity of each.

(b) Even if not considered mistakes, what changes would you make if you could do it all over again?

(c) What have been the biggest changes made to the language (albeit probably by other people) since its early development? Were these changes or new capabilities considered originally and dropped in the initial development, or were they truly later thoughts?

(d) Have changes been suggested but not adopted? If so, be as explicit as possible about changes suggested, and why they were not adopted.

4. Problems

(a) What were the biggest problems you had during the language design process? Did these affect the end result significantly?

(b) What are the biggest problems the users have had?

(c) What are the biggest problems the implementers have had? Were these deliberate, in the sense that a conscious decision was made to do something in the language design, even if it made the implementation more difficult?

(d) What trade-offs did you consciously make during the language design process? What trade-offs did you unconsciously make?

(e) What compromises did you have to make to meet other constraints such as time, budget, user demands, political, or other factors?

IV. IMPLICATIONS FOR CURRENT AND FUTURE LANGUAGES

1. Direct Influence

(a) What language developments of today and the foreseeable future are being directly influenced by your language? Regardless of whether your answer is “none” or “many, such as…,” please indicate the reasons.

(b) Is there anything in the experience of your language development which should influence current and future languages? If so, what is it? Put another way, in light of your experience, do you have advice for current and future language designers?

(c) Does your language have a long-range future? Regardless of whether your answer is “yes” or “no”, please indicate the reasons.

2. Indirect Influence

(a) Are there indirect influences which your language is having now? Are there any indirect influences that it can be expected to have in the near future? What are these, and why do you think they will be influential?
QUESTIONS ON THE EVOLUTION OF A PROGRAMMING LANGUAGE

The principle objective of a contribution in this category is to treat the history of a major language subsequent to its original development. In many cases, this entails extending the history of some language whose origins were treated in HOPL-I, or in HOPL-II. (Authors of papers in this category will be working with a member of the Program Committee, who will keep you informed of other relevant papers under consideration.)

When a programming language is first developed, it is typically the work of an individual or a small, concentrated group. Later development of the language is often the result of an expanded, re-staffed group, and perhaps additional individuals or groups outside the original organization. Similarly, while the original work is often focused on language design and implementation for a single environment, later developments are undertaken in a broader arena.

When compared with questions about the early history of a language, the following questions reflect this change in context. In particular, these questions are about the set of diverse development activities that surround the language, such as standardization, new implementations, significant publications, language-oriented groups (e.g., SIGs, user groups), etc.

These questions are grouped into the same four broad categories that apply to papers on the origins of a language:

* Background
* Rationale
* A posteriori evaluation
* Implications for current and future languages

The first question applies broadly to the language itself. It is intended to identify the particular development activities that are the focus of the history paper. In contrast, you should address the remaining questions for each of the activities identified in that initial background section.

You might also have significant information to contribute in response to questions raised about the early history of the language. Please examine the questions provided for authors of “early history” papers.

Where appropriate, your contribution should make reference to related papers from HOPL-I, HOPL-II, or HOPL-III.

I. BACKGROUND

1. Basic Facts About Activities, Organizations and People

(a) What are the categories of development to be discussed (e.g., standardization, new implementations, open source, significant publications) and what specific activities are reported on?

(From this point on, each question is intended to apply independently to each development activity identified in question I.1(a) above.)

(b) What organizations played principal roles in these developments? Identify them as precisely as possible: corporation and division, university and department, agency and office, etc. How were these organizations sponsored and funded?

(c) What, if any, was the nature of the cooperation or competition among these organizations?

(d) Who were the people involved in these developments? How were they related organizationally to each other and to the original developers for this language? Please be as specific as possible regarding names, titles, and dates.

(e) How did the roles of various individuals change during the course of the activity?
2. Costs and Schedules
(a) What was the source and amount of funding for supporting the development? Was it adequate?
(b) What was the schedule, if any?
(c) What was the estimated human effort required to carry it out?
(d) What was the estimated cost of the development?
(e) What were the effects of cost and schedule constraints?

3. Basic Facts About Documentation
(a) What are the significant publications arising from development? For each provide:
   • A specific reference
   • Names of authors (if not part of the reference)
   • Intended audience (e.g., user, implementer)
   • Format (e.g., manual, general trade book, standards document)
   • Availability

4. Languages/Systems Known at the Time
(a) What languages or systems other than the one in question had an effect on the development? In what ways did they affect the development?
(b) How did information about each of these languages or systems become available to the people it influenced?

5. Intended Purposes and Users
(a) What was the intended purpose of the development?
(b) Were the results proprietary, for sale, freely distributed, etc.?
(c) For whom were its results intended? How did this group of people differ from the originally intended set of users for this language?

6. Motivation
(a) Who was the prime mover for the development?
(b) What was the underlying motivation for the development?

II. RATIONALE FOR THE CONTENT OF THE DEVELOPMENT
To the extent that is appropriate, apply the “early history” questions in each of the following subcategories to the activity being addressed.

1. Environment Factors
(a) What were the effects on the development of concerns about program size, program libraries, portability, user background, execution efficiency, target computer architecture and speed, compilation environment, programming ease, execution environment, character set, parallel implementation, standardization, networked or parallel environment?
(b) In what ways had the environment changed since the original development of the language?
Final Program

2. Expected Applications of the Language
(a) How did expected applications influence choice of operations, data types, and objects?
(b) What features were essential to meet intended applications?

3. Design Principles Applied to the Development
(a) What, if any, was the underlying, consciously applied design philosophy?
(b) What considerations were made for detecting and correcting errors in the development?
(c) What role did “simplicity” play, and what was meant by “simplicity”?
(d) What role did “understandability” play? Which was given higher priority: “ease of reading” or “ease of writing”, and why?
(e) Were certain aspects of language (e.g., data types, operations, objects) considered more fundamental to the development than others? Why?

4. Language Definition
(a) What language definition techniques were used in this development? To what extent was the result of the development influenced by these choices?

5. Concepts of Other Languages
(a) To what extent was the introduction of new language concepts or features a part of this development?
(b) In what ways did concepts from other languages influence this development?

6. Influence of Non-Technical Factors
(a) What was the effect of other, similar developments on this one (e.g., overlapping standardization efforts)?
(b) What was the effect of time and cost constraints on the development?
(c) How did the size and structure of the development group affect results?

III. A POSTERIORI EVALUATION
1. Meeting of Objectives
(a) How well did the development meet its objectives?
(b) How well did the users feel the development met its objectives?
(c) What was the reaction of the computing community at large?
(d) How did portability of results impact their acceptance?
(e) Did the objectives of the development change over time? If so, when, how, and why did they change?

2. Contributions of the Development
(a) What were the biggest contributions of this development? Were they among the original objectives?
Final Program

(b) What do you consider its best features? What do you consider its worst features?

(c) How has this development affected other activities (e.g., development of other languages, dialects, language processors, standards, operating systems, and computer hardware)? Which of these other activities have become significant in their own right?

(d) In what way is the computer field better or worse off because of this development?

(e) What fundamental effects on programming language methodology have arisen from this development? (e.g., new data types, control structures, techniques for definition, for types of documentation, application design strategies, theoretical discoveries, etc.)

3. Mistakes or Desired Changes

(a) What mistakes were made in the development? Were these mistakes corrected in later developments?

(b) What changes would you now make, if you could?

(c) What were the biggest changes made to the development results since they were first released? Had they been considered earlier and then dropped, or were they truly later thoughts?

(d) What significant changes have been suggested but not adopted, and why?

4. Problems

(a) What were the major obstacles in carrying out the development?

(b) What were the major problems encountered by people who used its results: e.g., language users, designers, implementors, standards committees, etc.?

(c) What trade-offs were made during the development? Which were made consciously and which were recognized after the fact?

(d) What compromises were made to meet other constraints such as time, budget, user demand, and political factors?

(e) Which estimates (time, cost, effort, and human resources) were farthest from reality? Why were they off?

(f) What application-specific features might better have been left out?

IV. IMPLICATIONS FOR CURRENT AND FUTURE LANGUAGES

(a) Which current and foreseeable developments are being directly or indirectly influenced by this development, and why?

(b) Which results, if any, of this development have a long range future? Why?