C and C++ are obsolete

5th March 2003

Abstract

The Object Orientedness, Rich APIs, Exception Handling, Garbage Collection and Automatic Type Checking features of languages like Java outweigh any performance penalties these languages may have. While C and C++ might live on as compilation targets, their days as the programming languages of choice are numbered. Each of these modern language features contribute to higher productivity and we argue that it is evolutionary for more productive languages to replace the less productive ones. We define language obsolescence, look at life-cycles of languages, where C and C++ are well into their second halfs, and identify newer languages at earlier stages of the lifecycle. We further identify how features of these new languages contribute to greater productivity - the only compelling reason to move from one preferred language to another.

1 Scope of this work

1.1 What is obsolescence of programming languages? And (when) do languages become obsolete?

Obsolescence is being of a kind or style no longer current. When a language ceases to be the choice for new development, or there are efforts to rewrite software written in it, in newer ones, or the only major effort expended using it is in maintaining legacy systems, it is obsolete. It is not measured by the number of people programming or maintaining it. The largest number of programmers today may be in Rexx and COBOL, mostly maintaining legacy systems but by the definition used here, they are obsolete. A language becomes obsolete when there evolve newer, more productive languages with the same functionality.

1.2 Productivity and the productivity argument

Productivity is a measure of the amount of work done in unit time. During development it could be the number of functions points[1] or lines of code implemented in unit time. Post development it could be number of man hours to detect and repair a bug.

1.3 Are C and C++ obsolete? And what are we setting out to do?

Not yet. Notwithstanding the title of the paper. We are setting out to prove that C and C++ will become obsolete in the not very distant future. Our argument is two fold. The first approach is evolutionary - when languages have lived through their life-cycle they are replaced by newer ones. The second approach is by showing through empirical evidence that less productive languages are replaced by more productive ones if they have equivalent functionality, and then showing that Java is more productive than C and C++.

1.4 On the distinction between C and C++ and the choice of Java

There are significant differences between C and C++, and we try to be precise when there are arguments relevant to only one of them. On the other hand we choose Java as representative of a new class of languages identified by object orientedness, rich

APIs, exception handling, garbage collection and type checking. It does differ from, say Python or C#, but we make our case using Java. This is sufficient because we need to show the existence of one major, newer language offering greater productivity with the same functionality to make our case. Perhaps Java itself will be rendered obsolete by C#. But that is independent of the obsolescence or otherwise of C/C++.

2 Counter Arguments and Counter Counter Arguments

2.1 There is no such thing as a theory of evolution of languages. And what is the basis of a productivity based theory?

Most of the major languages have survived. Also, there are languages such as SmallTalk with higher productivity rates, yet used in a scale much smaller to less productive languages. While there have been abberations, the trend for major languages has follwed the increasing productivity path. From its scale of use we can already see that Java is not going the SmallTalk way. See the appendix and [3]for an extensive list of languages along with their productivity averages as measured in number of function points per month. Particularly for major languages the trend when going from ASSEMBLY to FORTRAN and COBOL, and from there to Pascal and C, and later to C++ and Java has been one of higher productivity. Note that the list includes some highly specialized languages like SQL, which are not general purpose programming languages.

2.2 There is no such thing as a language lifecycle. There may exist eternal languages

Again, most of the major languages have survived. There have been new versions of languages like FORTRAN, COBOL and Ada in the last ten years. They continue to find pockets of use. But if sometime in the next decade C and C++ are in the position FORTRAN, COBOL, and their likes are in now, then we would have won our argument. By our definition of obsolescence FORTRAN, COBOL and their likes are obsolete. Not all languages necessarily exhibit all phases of the lifecycle, but the older languages surviving today are either in Deprecated or Decayed state(see section 3).

2.3 There is a huge body of C/C++ software and C/C++ expertise. It may be more productive to have people continue using them.

There is a learning curve associated with learning new languages, and loss in productivity when existing expertise has to be set aside. But this is true in the short to medium term only. And continuance of C and C++ in legacy systems and programmers to maintain them does not save these languages from becoming obsolete.

2.4 For a certain class of applications, it may be absolutely necessary to program in C

For Operating Systems, device drivers and other systems level programming you will need C. But this is an increasingly small part of the software universe. Also, it may be better to program them largely in Java then translate that to C code (hence the role of C as a target of compilation). We thereby derive productivity benefits of Java even when the application requires C.

3 Programming Language Lifecycle

For several programming languages, there exists a lifecycle along the following general pattern[5]

1. Conception: Certain things are missing in existing languages, they can not be done satisfactorily.

2. Adoption: The new idea is powerful and there are enough APIs to make ports tractible.

3. Acceptance: Brought forth by stability and early success. But there are things missing in terms of flexibility, applicability, development environments, tools, etc.

4. Maturation: Functionality over form. We begin to lose track of why the language was created to begin with. Changes are made to make the language more useful but less portable. Standards are developed to try to lock down feature sets to ensure portability.

5. Inefficiency: When function has won, at the cost of elegance and efficiency. Libraries become cumbersome as they build upon one another and the language itself becomes subordinated to the library structure.

6. Deprecation: The language is now too heavy and cumbersome.

7. Decay: Still in use, but most likely, only for maintenance. New releases stop.

Not all languages necessarily exhibit all phases, but the major ones go through most. Phases 1 through 4 are marked by increasing productivity, and after stage 5 there is decline. The timeline and development history of major programming languages; Assembly(1956), FORTRAN(1957, there were 40 different FORTRAN compilers by 1963), COBOL(1959), C(1972, changed in 1978-79 and standardised by ANSI in the 1980s), C++(1983, the STL was developed in 1994), Java(1995, several classes added in each new version), indicates that the cycle exists and is between ten and thirty years.

4 Language Productivity

We propose evaluating language features for productivity along three dimensions.

4.1 Ability to model the problem space and model the solution space

The role of a program is to transform one data structure to another, ususally with a loss of information but increase in organization. The role of a programming language is to make that transformation easier. The closer we can get to modeling the real world the better, since we have less translation error. Machine language represents one extreme where everything is modeled as a bit sequence. Languages like FORTRAN were better as they atleast used floats and booleans. The second notion, getting closer to the solution space is essentially the same matter. For example, languages that let us directly represent and manage communication protocols are easier to use for internet messaging than languages that force us to program controls as a series of bits written to a port. The closer languages are to problem and solution space, the more intuitive, and easier it is to learn and program in them.

4.2 Ability to reuse code and the existence of a rich Application Programming Interface

Reuse is a classical software engineering principle. It results in reduced time for testing, service and maintenance. It is also easier to introduce new functionality. Part of the power of object oriented systems is the promise of code reuse. An extensive API is also essentially for code reuse. With common algorithms and data structures already implemented, the programmer has more time to solve the specific problem.

4.3 Low probability of bugs and low mean time to repair bugs

Lower probability of bugs means lesser time spent testing and debugging. Low mean time to repair means lower time to diagnose and debug. Certain language features - what languages allow, and equally important what languages prohibit, play an important role in bug rate and repair time.

5 The Arguments:

5.1 The lifecycle argument

5.1.1 C is into Inefficiency and bordering on Deprecation. C++ is getting to the end of Maturation

This follows from the definition. C was standardised by ANSI in the 1980s (maturation stage) and through developments in the 1990s, the language is now largely governed by its library structure. C++ has a more recent STL, to which there are periodc extensions.

5.1.2 Java has been Accepted and is beginning to get Mature

It has enjoyed early success. There are new classes added, and methods revised and updated with each new release. It was originally conceived for digitally controlled consumer devices but has long since outgrown that. The evolution argument is general, and doesn't give direct technical or economic reasons for change. But it does give a measure of the rate of change. We may use it to predict the time to obsolescence of a programming language. There are other languages too, close to Maturation like Python and C#.

5.2 The productivity argument

In each of the three productivity dimensions Java does better than C and C++.

5.2.1 Modeling problem and solution spaces

Object oriented languages like Java provide powerful features to model the real world. An object is an encapsulation of data and operations on that data. This is widely believed to be an improvement over the procedural model of languages like C. Its also closer to the solution space. The Java thread library is closer to distributed design space, its graphic library (Swing) is closer to user interface design space, and its networking library is closer to networking design space, to name a few.

5.2.2 Code reuse and a rich API

Modelling using objects can keep dependency between parts of a system low, allowing systems to be extended and reused. JDK 1.4 has over 3700 public classes.

5.2.3 Low bug probability and mean time to repair

Shipping C code has, on average, one bug per 55 lines of code. About half of these bugs are related to memory allocation and deallocation. In a study comparing bug and productivity rates in Java and C++[2], C++ programs were found to have two to three times as many bugs per line of code as Java, between 15 and 50 percent more defects and took six times as long to debug. Defects are errors detected during compile time and bugs are problems detected during testing and deployment. In the same study Java was found to be between 30 and 200 percent more productive in terms of lines of code per minute. The metrics for lines of code can be extended to function points. Functionality is commonly measured in terms of function points and program size is some function of functionality. If the same functionality is implemented using Java and C++ then it is reasonable to assume that the two will be similar in size because the languages have very similar expressive power. Exception handling, garbage collection and type safety contribute to lower bug probability and mean time to repair

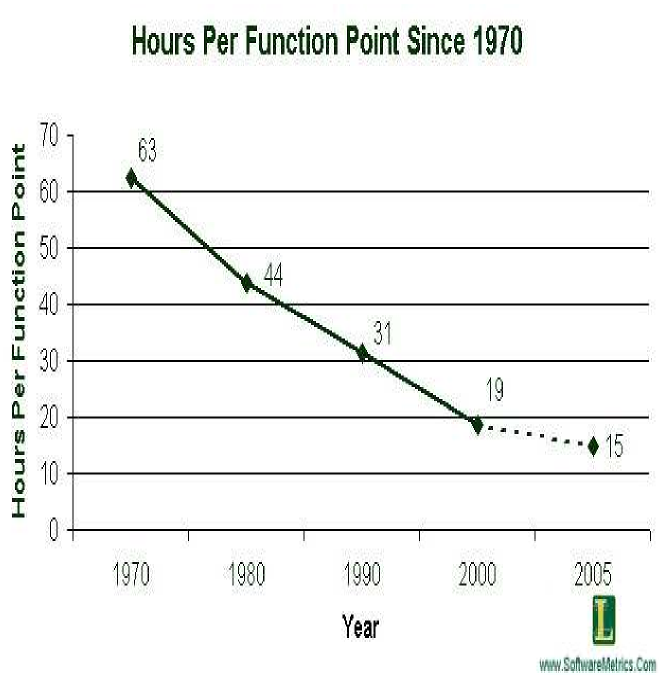
Exception handling: Exception handling features separate program logic from exception logic, classify exception types and present a common model for reporting error conditions. They make it easier to track the origin of errors reducing mean time to recovery.

Garbage Collection: This is the process of automatically freeing objects that are no longer referenced by the program. This frees the programmer from having to keep track of when to free allocated memory, thereby preventing many potential bugs. Also, knowing when to explicitly free allocated memory is not easy. Giving this job to the Java VIrtual Machine(JVM) means the programmer can focus on real product development. A second advantage of garbage collection is that it helps ensure program integrity. Garbage collection is an important part of Java's security strategy. Java programmers are unable to accidentally or intentionally crash the JVM by incorrectly freeing memory. Good garbage collection algorithms have been developed, and adequate performance can be achieved for all but the most demanding of applications.

Type Safety: The Java language is designed to enforce type safety. This means that programs are prevented from accessing memory in inappropriate ways. More specifically, every piece of memory is part of some Java object. Every Java object is stored in some region of the computer's memory. Java labels every object by putting its class tag next to it. Dynamic type checking is done by checking the class tag of the object before every operation on the object. To improve performance, Java uses static type checking too. The type checker looks at the program before it is run and determines which way the tag checking operations will come out, in advance, so these may be omitted at run time. This is more efficient but also more complicated than dynamic type checking. The Java language and byte code formats are crafted to facilitate static type checking. The byte code Verifier is a static type checker, eliminating many tag checking operations from Java programs. The result is a type safe program that runs efficiently. Static type checking has other advantages, too. For example, static type checking can be done at compile time, thus informing the software developer of any type errors before the code is shipped.

5.3 Why higher productivity is still required.

The productivity of software projects has increased about three fold since 1970[4]. In 1970, COBOL was the state of the art. By 2000, developers had started the using languages like Java and productivity increased by a third in the 1990s alone[See figure 1(a)]. But the size of software projects has increased ten fold during the same time, or more than three times faster. [See figure 1(b)]. And the trend is towards larger and more complex software. To keep the num-



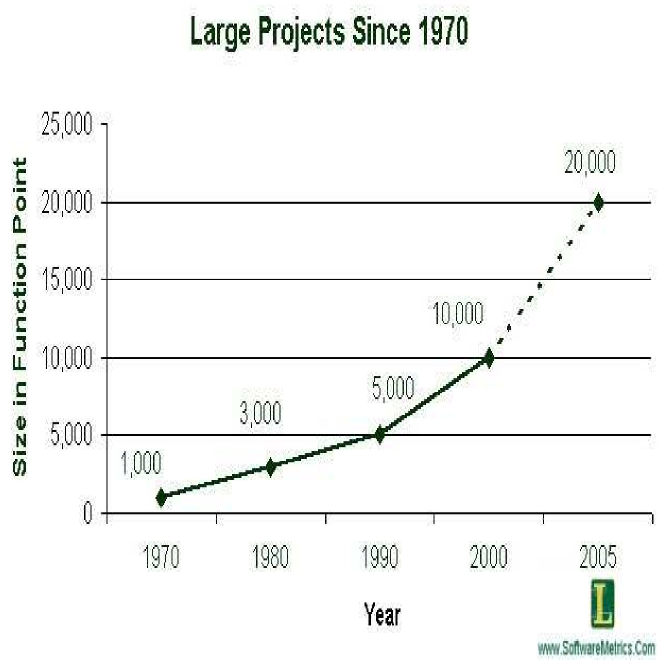


Figure 1: (a) and (b)

ber of man hours requied for development constant, productivity will have to **double** by 2005.

6 Conclusion

Programming languages have a lifecycle and are periodically replaced by newer ones. Certain languages persist longer than others. C and C++ are in the second half of their lifecycle, and there are languages like Java more towards the beginning of theirs. A language is replaced as the preferred language of choice when it is more productive to use another provided the other language supports all or most of the operations supported by the current. There is empirical evidence in this regard, and Java (and similar languages) offers better productivity than than C and C++ with nearly the same functionality.

References

[1] Function Point Counting Practices Manual. International Function Point Users Group Standards. January 1994

[2] Comparing Observed Bug and Productivity Rates for Java and C++. Geoffrey Phipps

[3] Programming Languages Table, Release 8.2 March 1996. Caper Jones. Software Productivity Research

[4] Productivity of Software from 1970 to present. Long Street Consulting. 2000

[5] Component Oriented Software, Java Perspectives. Robin Sharp 2002

[6] An Introduction to the Garbage Collected Heap of the Java Virtual Machine. Bill Venners. Java World August 1996

[7] Securing Java. Gary McGraw, Ed Felten. 1999

[8] C++ Vs. Java. Ray Cox. Shadowlink.net. 1998

[9] The Essence of Java. Bruce Eckel. 2000

7 Appendix

Programming Languages Table

As language levels go up, fewer statements to code one Function Point are required. For example, COBOL is a level 3 and requires about 105 statements per Function Point.

The numeric levels of various languages provide a convenient shortcut for converting size from one language to another. For example, if an application requires 1000 non-commentary COBOL statements (level 3), then it would take only 500 statements in a level 6 language (such as NATURAL) and only 250 statements in a level 12 language (such as OBJECTIVE C). As you can see, the average number of statements required is proportional to the levels of the various languages.

The correlation between the level of a language and development productivity is not linear. For most large software projects, coding amounts to only about 30 percent of the effort.

Table 1. Language Level Relationship to Productivity

LANGUAGE LEVEL	PRODUCTIVITY AVERAGE PER STAFF MONTH
1 - 3	5 to 10 Function Points
4 - 8	10 to 20 Function Points
9 - 15	16 to 23 Function Points
16 - 23	15 to 30 Function Points
24 - 55	30 to 50 Function Points
Above 55	40 to 100 Function Points

What Is The Basis For Language Levels?

The languages and levels in Table 2 were gathered in four ways.

* Counting Function Points and Source Code
* Counting Source Code
* Inspecting Source Code
* Researching Languages

Counting Function Points And Source Code

Actual counts of Function Points and source code statements were performed. Samples of counting Function Points and source code statements were done on Ada, several BASIC dialects, COBOL, PASCAL, and PL/I.

Counting Source Code

Source code statements were counted, then compared to the size of the same program in languages of known levels. Assembly, APL, C, OBJECTIVE C, FORTH, FORTRAN, LISP, PILOT, and PROLOG are languages that produce the same source code count as COBOL. So code sizes were compared to the known quantity of COBOL source code.

Inspecting Source Code

Source code inspection for common applications was done. Then the volume of code for the application in a measured language was hypothesized. ACTOR, CLARION, and TRUE BASIC are examples of languages that were inspected and their levels hypothesized by subjective means.

Researching Languages

Research was done by reading descriptions and genealogies of languages and making an educated guess as to their levels. KL, CLOS, TWAICE, and FASBOL are examples of languages that were assigned tentative levels merely from descriptions of the language, rather than from actual counts.

For spreadsheets the ordinary concepts of a language do not apply. In this case, formulas, labels, and constants were considered to be statements.

Table 2. Programming Languages and Levels

LANGUAGE	LEVEL	AVERAGE SOURCE STATEMENTS PER FUNCTION POINT
1032/AF	20.00	16
1st Generation default	1.00	320
2nd Generation default	3.00	107
3rd Generation default	4.00	80
4th Generation default	16.00	20
5th Generation default	70.00	5
AAS Macro	3.50	91
ABAP/4	20.00	16
ACCEL	17.00	19
Access	8.50	38
ACTOR	15.00	21
Acumen	11.50	28
Ada 83	4.50	71
Ada 95	6.50	49
ADR/DL	8.00	40
ADR/IDEAL/PDL	16.00	20
ADS/Batch	16.00	20
ADS/Online	16.00	20
AI shell default	6.50	49
AI SHELLS	6.50	49
ALGOL 68	3.00	107
ALGOL W	3.00	107
AMBUSH	10.00	32
AML	6.50	49
AMPPL II	5.00	64

ANSI BASIC	5.00	64
ANSI COBOL 74	3.00	107
ANSI COBOL 85	3.50	91
ANSI SQL	25.00	13
ANSWER/DB	25.00	13
APL 360/370	10.00	32
APL default	10.00	32
APL*PLUS	10.00	32
APPLESOFT BASIC	2.50	128
Application Builder	16.00	20
Application Manager	9.00	36
APS	19.00	17
APT	4.50	71
APTools	16.00	20
ARC	6.50	49
Ariel	3.00	107
ARITY	6.50	49
Arity PROLOG	5.00	64
ART	6.50	49
ART-IM	7.00	46
ART Enterprise	7.00	46
Artemis	8.00	40
AS/SET	17.00	19
ASI/INQUIRY	25.00	13
ASK Windows	7.00	46
Assembly (Basic)	1.00	320
Assembly (Macro)	1.50	213
Associative default	5.00	64
Autocoder	1.00	320
awk	15.00	21
Aztec C	2.50	128
BALM	3.00	107
BASE SAS	6.00	53
BASIC	3.00	107
BASIC A	2.50	128
Basic assembly	1.00	320
Berkeley PASCAL	3.50	91
BETTER BASIC	3.50	91
BLISS	3.00	107
BMSGEN	9.00	36
BOEINGCALC	50.00	6
BTEQ	25.00	13
C	2.50	128
C Set 2	3.50	91
C++	6.00	53
C86Plus	2.50	128
CA-dBFast	8.00	40
CA-EARL	11.50	28
CAST	6.50	49
CBASIC	3.50	91
CDADL	16.00	20
CELLSIM	7.00	46
Centerline C++	6.00	53
CHILI	3.00	107
CHILL	3.00	107
CICS	7.00	46
CLARION	5.50	58
CLASCAL	4.00	80
CLI	10.00	32
CLIPPER	17.00	19
CLIPPER DB	8.00	40
CLOS	15.00	21
CLOUT	8.00	40
CMS2	3.00	107
CMSGEN	17.00	19
COBOL	3.00	107
COBOL II	3.00	107
Cobol/400	3.50	91
COBRA	16.00	20
CodeCenter	9.00	36
Cofac	9.00	36
COGEN	9.00	36
COGNOS	9.00	36
COGO	4.50	71
COMAL	4.00	80
COMIT II	5.00	64
Common LISP	5.00	64
Concurrent PASCAL	4.00	80
CONNIVER	5.00	64
CORAL 66	3.00	107
CORVET	17.00	19
CorVision	22.00	15
CPL	2.00	160
Crystal Reports	16.00	20
CSL	6.50	49
CSP	6.00	53
CSSL	7.00	46
CULPRIT	25.00	13
CxPERT	6.50	49
CYGNET	17.00	19
Data base default	8.00	40
Dataflex	8.00	40

Datatrieve	16.00	20
dBase III	8.00	40
dBase IV	9.00	36
DCL	1.50	213
DEC-RALLY	8.00	40
Decision support default	9.00	36
DELPHI	11.00	29
DL/1	8.00	40
DNA-4	17.00	19
DOS Batch Files	2.50	128
DSP Assembly	2.00	160
DTABL	7.00	46
DTIPT	7.00	46
DYANA	4.50	71
DYNAMO-III	7.00	46
EASEL	11.00	29
EASY	6.50	49
EASYTRIEVE +	25.00	13
Eclipse	6.50	49
ED-Scheme 3.4	6.00	53
EDA/SQL	27.00	12
EIFFEL	15.00	21
ENFORM	7.00	46
English-based default	6.00	53
Ensemble	11.00	29
EPOS	16.00	20
Erlang	8.00	40
ESF	8.00	40
ESPADVISOR	6.50	49
ESPL/I	4.50	71
EUCLID	3.00	107
EXCEL 1-2	51.00	6
EXCEL 3-4	55.00	6
EXCEL 5	57.00	6
EXPRESS	9.00	36
EXSYS	6.50	49
Extended Common LISP	5.75	56
EZNOMAD	9.00	36
Facets	16.00	20
FactoryLink IV	11.00	29
FAME	9.00	36
FileMaker Pro	9.00	36
FLAVORS	11.00	29
FLEX	7.00	46
FlexGen	11.00	29
FOCUS	8.00	40
FOIL	6.00	53
Forte	18.00	18
FORTH	5.00	64
FORTRAN 66	2.50	128
FORTRAN 77	3.00	107
FORTRAN 90	4.00	80
FORTRAN 95	4.50	71
FORTRAN	3.00	107
FORTRAN II	2.50	128
Foundation	11.00	29
FOXPRO 1	8.00	40
FOXPRO 2.5	9.50	34
FRAMEWORK	50.00	6
G2	6.50	49
GAMMA	20.00	16
Genascript	12.00	27
GENER/OL	25.00	13
GENEXUS	21.00	15
GENIFER	17.00	19
GeODE 2.0	20.00	16
GFA Basic	9.50	34
GML	7.00	46
Golden Common LISP	5.00	64
GPSS	7.00	46
GUEST	11.50	28
Guru	6.50	49
GW BASIC	3.25	98
Haskell	8.50	38
High C	2.50	128
HLEVEL	5.50	58
HP BASIC	2.50	128
HTML 2.0	20.00	16
HTML 3.0	22.00	15
Huron	20.00	16
IBM ADF I	16.00	20
IBM ADF II	18.00	18
IBM Advanced BASIC	3.25	98
IBM CICS/VS	8.00	40
IBM Compiled BASIC	3.50	91
IBM VS COBOL	3.00	107
IBM VS COBOL II	3.50	91
ICES	4.50	71
ICON	4.00	80
IDMS	8.00	40
IEF	23.00	14
IEW	23.00	14

IFPS/PLUS	10.00	32
IMPRS	8.00	40
INFORMIX	8.00	40
INGRES	8.00	40
INQUIRE	25.00	13
INSIGHT2	6.50	49
INSTALL/1	20.00	16
INTELLECT	6.00	53
INTERLISP	5.50	58
Interpreted BASIC	3.00	107
Interpreted C	2.50	128
IQLISP	5.50	58
IQRP	25.00	13
JANUS	4.50	71
JAVA	6.00	53
JCL	1.45	221
JOSS	3.00	107
JOVIAL	3.00	107
KAPPA	8.00	40
KBMS	6.50	49
KCL	5.00	64
KEE	6.50	49
Keyplus	8.00	40
KL	5.00	64
KLO	5.00	64
KNOWOL	6.50	49
KRL	5.50	58
KSH	15.00	21
Ladder Logic	9.00	36
LAMBIT/L	5.00	64
Lattice C	2.50	128
Liana	2.50	128
LILITH	4.50	71
LINC II	23.00	14
LISP	5.00	64
LOGLISP	5.50	58
LOOPS	15.00	21
LOTUS 123 DOS	50.00	6
LOTUS Macros	3.00	107
LUCID 3D	51.00	6
LYRIC	6.00	53
M	20.00	16
macFORTH	5.00	64
MACH1	8.00	40
Machine language	0.50	640
Macro assembly	1.50	213
MAESTRO	20.00	16
MAGEC	20.00	16
MAGIK	15.00	21
MAKE	15.00	21
MANTIS	8.00	40
MAPPER	6.00	53
MARK IV	8.00	40
MARK V	9.00	36
MATHCAD	60.00	5
MDL	9.00	36
MENTOR	6.00	53
MESA	3.00	107
Microfocus COBOL	4.00	80
microFORTH	5.00	64
Microsoft C	2.50	128
MicroStep	16.00	20
Miranda	8.00	40
Model 204	8.50	38
MODULA 2	4.00	80
MOSAIC	50.00	6
MS C ++ V. 7	6.00	53
MS Compiled BASIC	3.50	91
MSL	5.00	64
muLISP	5.00	64
MUMPS	17.00	19
NASTRAN	4.50	71
NATURAL 1	6.00	53
NATURAL 2	7.00	46
NATURAL Construct	13.00	25
Natural language	0.10	3200
NETRON/CAP	17.00	19
NEXPERT	6.50	49
NIAL	6.50	49
NOMAD2	8.00	40
Non-procedural default	9.00	36
Notes VIP	9.00	36
Nroff	6.00	53
Object-Oriented default	11.00	29
OBJECT Assembler	5.00	64
Object LISP	11.00	29
Object LOGO	11.00	29
Object PASCAL	11.00	29
Object Star	20.00	16
Objective-C	12.00	27
ObjectVIEW	13.00	25
OGL	4.00	80

OMNIS 7	8.00	40	QUATTRO PRO	51.00	6
OODL	11.00	29	Query default	25.00	13
OPS	7.00	46	QUICK BASIC 1	5.00	64
OPS5	5.50	58	QUICK BASIC 2	5.25	61
ORACLE	8.00	40	QUICK BASIC 3	5.50	58
Oracle Developer/2000	14.00	23	Quick C	2.50	128
Oscar	3.00	107	Quickbuild	11.50	28
PACBASE	22.00	15	QUIZ	22.00	15
PACE	8.00	40	RALLY	8.00	40
PARADOX/PAL	9.00	36	RAMIS II	8.00	40
PASCAL	3.50	91	RapidGen	11.50	28
PC FOCUS	9.00	36	RATFOR	3.50	91
PDL Millenium	15.00	21	RDB	8.00	40
PDP-11 ADE	6.00	53	REALIA	7.00	46
PERL	15.00	21	Realizer 1.0	8.00	40
Persistance Object Builder	15.00	21	Realizer 2.0	9.00	36
PILOT	6.00	53	RELATE/3000	8.00	40
PL/I	4.00	80	Reuse default	60.00	5
PL/M	4.50	71	REXX (MVS)	4.00	80
PL/S	3.50	91	REXX (OS/2)	7.00	46
PLANIT	6.00	53	RM BASIC	3.50	91
PLANNER	5.00	64	RM COBOL	3.00	107
PLANPERFECT 1	45.00	7	RM FORTRAN	3.00	107
PLATO	6.00	53	RPG I	4.00	80
polyFORTH	5.00	64	RPG II	5.50	58
POP	5.50	58	RPG III	5.75	56
POPLOG	5.50	58	RT-Expert 1.4	5.50	58
Power BASIC	6.50	49	S-PLUS	10.00	32
PowerBuilder	20.00	16	SAIL	3.00	107
POWERHOUSE	23.00	14	SAPIENS	20.00	16
PPL (Plus)	8.00	40	SAS	10.00	32
Pro-C	12.00	27	SAVVY	25.00	13
PRO-IV	5.50	58	SBASIC	3.50	91
Problem-oriented default	4.50	71	SCEPTRE	4.50	71
Procedural default	3.00	107	SCHEME	6.00	53
Professional PASCAL	3.50	91	Screen painter default	57.00	6
Program Generator default	20.00	16	SEQUAL	27.00	12
PROGRESS V4	9.00	36	SHELL	15.00	21
PROLOG	5.00	64	SIMPLAN	9.00	36
PROSE	3.00	107	SIMSCRIPT	7.00	46
PROTEUS	3.00	107	SIMULA	7.00	46
QBasic	5.50	58	SIMULA 67	7.00	46
QBE	25.00	13	Simulation default	7.00	46
QMF	22.00	15	SMALLTALK 286	15.00	21
QNIAL	6.50	49	SMALLTALK 80	15.00	21
QUATTRO	51.00	6	SMALLTALK/V	15.00	21

SNAP	4.00	80	TUTOR	6.00	53
SNOBOL2-4	2.50	128	TWAICE	6.50	49
SoftScreen	23.00	14	UCSD PASCAL	3.50	91
SOLO	5.50	58	UFO/IMS	9.00	36
SPEAKEASY	9.00	36	UHELP	10.00	32
Spinnaker PPL	9.00	36	UNIFACE	20.00	16
Spreadsheet default	50.00	6	UNIX Shell Scripts	15.00	21
SPS	1.00	320	VAX ACMS	5.50	58
SPSS	10.00	32	VAX ADE	8.00	40
SQL	25.00	13	VECTRAN	3.00	107
SQL-Windows	27.00	12	VHDL	17.00	19
Statistical default	10.00	32	Visible C	6.50	49
STRATEGEM	9.00	36	Visible COBOL	8.00	40
STRESS	4.50	71	Visicalc 1	35.00	9
Strongly typed default	3.50	91	Visual 4.0	11.00	29
STYLE	7.00	46	Visual Basic 1	7.00	46
SUPERBASE 1.3	9.00	36	Visual Basic 2	7.50	43
SURPASS	50.00	6	Visual Basic 3	8.00	40
SYBASE	8.00	40	Visual Basic 4	9.00	36
Symantec C++	11.00	29	Visual Basic 5	11.00	29
SYMBOLANG	5.00	64	Visual Basic DOS	8.00	40
Synchroworks	18.00	18	Visual C++	9.50	34
SYNON/2E	17.00	19	Visual COBOL	16.00	20
System-W	9.00	36	Visual Objects	20.00	16
Tandem Access Language	3.50	91	VisualAge	15.00	21
TCL	5.00	64	VisualGen	18.00	18
TELON	20.00	16	VS-REXX	10.00	32
TESSARACT	8.00	40	VULCAN	5.00	64
THE TWIN	50.00	6	VZ Programmer	9.00	36
THEMIS	25.00	13	WARP X	8.00	40
TI-IEF	23.00	14	WATCOM C	2.50	128
Topspeed C ++	11.00	29	WATCOM C/386	2.50	128
TRANSFORM	22.00	15	Waterloo C	2.50	128
TRANSLISP PLUS	5.75	56	Waterloo PASCAL	3.50	91
TREET	5.00	64	WATFIV	3.75	85
TREETRAN	5.00	64	WATFOR	3.50	91
TRS80 BASIC II,III	2.50	128	WHIP	3.50	91
TRUE BASIC	5.00	64	Wizard	11.50	28
Turbo C	2.50	128	XLISP	5.00	64
TURBO C++	6.00	53	YACC	6.00	53
TURBO EXPERT	6.50	49	YACC++	6.00	53
Turbo PASCAL >5	6.50	49	ZBASIC	3.50	91
Turbo PASCAL 1-4	4.00	80	ZIM	17.00	19
Turbo PASCAL 4-5	4.50	71	ZLISP	5.00	64
Turbo PROLOG	4.00	80			
TURING	4.00	80			