Growing a Language

Guy L. Steele Jr.
Sun Microsystems Laboratories
guy.steele@east.sun.com

October 22, 1998

Most of the time, I do not read my talks; I plan the main points and then speak off the top of my head. For this talk, I need to stick to a text, and for good cause. Please bear with me.

I think you know what a man is. A *woman* is more or less like a man, but not of the same sex. (This may seem like a strange thing for me to start with, but soon you will see why.)

Next, I shall say that a *person* is a woman or a man (young or old).

To keep things short, when I say "he" I mean "he or she," and when I say "his" I mean "his or her."

A *machine* is a thing that can do a task with no help, or not much help, from a person.

(As a rule, we can speak of two or more of a thing if we add an "s" or "z" sound to the end of a word that names it.)

$$
\begin{array}{lll}
\langle \text{noun} \rangle & ::= & \langle \text{noun that names one thing} \rangle \text{ "s"} \\
 & | & \langle \text{noun that names one thing} \rangle \text{ "es"}
\end{array}
$$

These are names of persons: *Alan Turing*, *Alonzo Church*, *Charles Kay Ogden*, *Christopher Alexander*, *Eric Raymond*, *Fred Brooks*, *John Horton Conway*, *James Gosling*, *Bill Joy*, and *Dick Gabriel*.

The word *other* means "not the same." The phrase *other than* means "not the same as."

A *number* may be nought, or may be one more than a number. In this way we have a set of numbers with no bound.

$$
\begin{array}{lll}
\langle \text{number} \rangle & ::= & \texttt{0} \\
 & | & \texttt{1 +} \ \langle \text{number} \rangle
\end{array}
$$

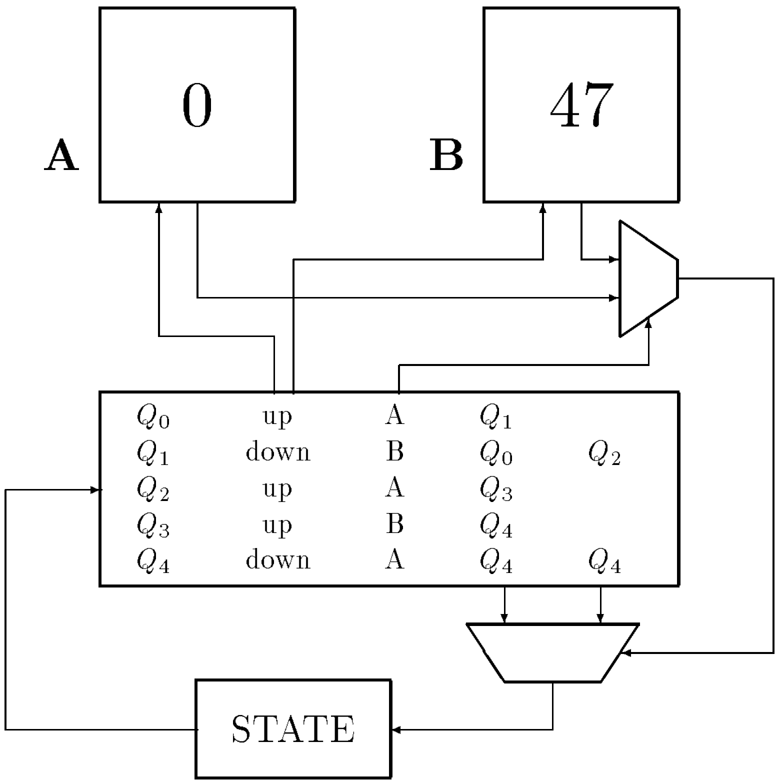
There are other numbers as well, but I shall not speak more of them yet.

These numbers—nought or one more than a number—can be used to count things. We can add two numbers if we count up from the first number while we count down from the number that is not the first till it come to nought; then the first count is the sum.

$$4 + 2 = 5 + 1 = 6 + 0 = 6$$

We shall take the word *many* to mean "more than two in number."

Think of a machine that can keep track of two numbers, and count each one up or down, and test if a number be nought and by such a test choose to do this or that. The list of things that it can do and of choices that it can make must be of a known size that is some number.



Here you can see the two numbers and a list. Each row in the list has a state name; the word "up" or "down"; and which number to count up or count down. For "up," we have the name of the next state to go to (and the machine counts the number up by one); for "down," the machine first tests the number, and so we have the name of the new state to go to if the number be nought and the name of the new state to go to if the number be other than nought (in which case the machine counts the number down by one).

A *computer* is a machine that can do at least what that machine can—and we have good cause to think that if a computer task can be done at all, then the two-number machine can do it, too, if you put numbers in and read them out in the right way. In some sense, all computers are the same; we know this thanks to the work of such persons as Alan Turing and Alonzo Church.

A *vocabulary* is a set of words.

A *language* is a vocabulary and rules for what a string of words might mean to a person or a machine that hears them.

(To *define* a word is to tell what it means. When you define a word, you add it to your vocabulary and to the vocabulary of each person who hears you. Then you can use the word.)

To *program* is to make up a list of things to do and choices to make, to be done by a computer. Such a list is called a *program.*

(A noun can be made from a verb to mean that which is done as meant by that verb; to make such a noun, we add "ing" to the verb stem. Thus we can speak of "programming.")

$\langle\text{noun}\rangle ::= \langle\text{verb stem}\rangle$ "ing"

A programming language is a language that we can use to tell a computer a program to do. *Java* is a brand name for a computer programming language. I shall speak of the Java programming language a great deal more in this talk. (I have to say the full phrase "Java programming language," for there is a guy who works where I do who deals with the laws of marks of trade, and he told me I have to say it that way.) Names of other programming languages are *Fortran*, *APL*, *Pascal*, and *PL/I*.

A part of a program that does define a new word for use in other parts of the program is called a *definition*. Not all programming languages have a way to code definitions, but most do. (Those that do not are for wimps.)

Should a programming language be small or large? A small programming language might take but a short time to learn. A large programming language may take a long, long time to learn, but then it is less hard to use, for we then have a lot of words at hand—or, I should say, at the tips of our tongues—to use at the drop of a hat. If we start with a small language, then in most cases we can not say much at the start. We must first define more words; then we can speak of the main thing that is on our mind.

(We can use a verb in the past tense if we add a "d" or "ed" sound at the end of the verb. In the same way we can form what we call a *past participle*, which is a form of the verb that says of a noun that what the verb means has been done to it.)

```
⟨verb⟩ ::=      ⟨verb stem⟩ "ed"
         |      ⟨verb stem⟩ "d"

⟨past participle⟩ ::=
                ⟨verb stem⟩ "ed"
          |     ⟨verb stem⟩ "d"
```

For this talk, I wanted to show you what it is like to use a language that is much too small. You have now had a good taste of it. I must next define a few more words.

An *example* is some one thing, out of a set of things, that I put in front of you so that you can see how some part of that thing is in fact a part of each thing in the set.

A *syllable* is a bit of sound that a mouth and tongue can say all at one time, more or less, in a smooth way. Each word is made up of one or more syllables.

A *primitive* is a word for which we can take it for granted that we all know what it means.

For this talk, I chose to take as my primitives all the words of one syllable, and no more, from the language I use for most of my speech each day, which is called *English*. My firm rule for this talk is that if I need to use a word of two or more syllables, I must first define it. I can do this by defining one word at a time—and I have done so—or I can give a rule by which a new word can be made from some other known word, so that many new words can be defined at once—and I have done this as well.

This choice, to start with just words of one syllable, was why I had to define the word "woman" but could take the word "man" for granted. I wanted to define "machine" in terms of the word "person", and it seemed least hard to define "person" as "a man or a woman"—but then, by chance, the word "man" has one syllable and so is a primitive, but the word "woman" has two syllables and so I had to define it. In a language other than English, the words that mean "man" and "woman" might each have one syllable—or might each have two syllables, in which case one would have to take some other tack.

We have to do this a lot when we write real computer programs: a thought that seems like a primitive in our minds turns out not to be a primitive in a programming language, and in each new program we must define it once more. A good example of this is `max`, which yields the more large of two numbers. It is a primitive thought to me, but few programming languages have it as a primitive; I have to define it. This is the sort of thing that makes a computer look like a person who is but four years old. Next to English, all computer programming languages are small; as we write code, we must stop now and then to define some new term that we will need to use in more than one place. Some persons find that their programs have a few large chunks of code that do the "real work" plus a large pile of small bits of code that define new words, so to speak, to be used as if they were primitives.

I hope that this talk brings home to you—in a way you can feel in your heart, not just think in your head—what it is like to have to program in that way. This should show you, true to life, what it is like to use a small language. Each time I have tried this sort of thing, I have found that I can not say much at all till I take the time to define at least a few new terms. In other words, if you want to get far at all with a small language, you must first add to the small language to make a language that is more large.

(In some cases we will find it more smooth to add the syllable "er" to the end of a word than to use the word "more" in front of it; in this way we might say "smoother" in place of "more smooth." Let me add that *better* means "more good.")

$$\langle\text{word that can change what a noun means}\rangle ::= \langle\text{word that can change what a noun means}\rangle \text{ "er"}$$

In truth, the words of one syllable form quite a rich vocabulary, with which you can say many things. You may note that this vocabulary is much larger than that of the language called *Basic English*, defined by Charles Kay Ogden in the year one-nine-three-nought. I chose not to use Basic English for this talk, the cause being that Basic English has many words of two or more syllables, some of them quite long, handed down to us from the dim past of Rome. While Basic English has fewer words, it does not give the feel, to one who speaks full English, of being a small language.

(By the way, from now on I shall use the word *because* to mean "the cause being that.")

If we look at English and then at programming languages, we see that all our programming languages seem small. And yet we can say, too, in some cases, that one programming language is smaller than some other programming language.

A *design* is a plan for how to build a thing. To design is to build a thing in one's mind but not yet in the real world—or, better yet, to plan how the real thing can be built.

The main thing that I want to ask in this talk is: If I want to help other persons to write all sorts of programs, should I design a small programming language or a large one?

I stand on this claim: I should not design a small language, and I should not design a large one. I need to design a language that can grow. I need to plan ways in which it might grow—but I need, too, to leave some choices so that other persons can make those choices at a later time.

This is not a new thought. We have known for some time that huge programs can not be coded from scratch all at once. There has to be a plan for growth. What we must stop to think on now is the fact that languages have now reached that large size where they can not be designed all at once, much less built all at once.

(Let me pause to define some names for numbers. *Twenty* is twice ten. *Thirty* is thrice ten. *Forty* is twice twenty. A *hundred* is ten times ten. A *million* is a hundred times a hundred times a hundred. *Sixteen* is twice eight. *Seven* is one plus six. *Fifty* is one more than seven squared. One more thing: *ago* means "in the past, counting back from now.")

In the past, it made sense to design a whole language, once for all. Fortran was a small language forty years ago, designed for tasks with numbers, and it served well. PL/I was thought a big language thirty years ago, but now we would think of it as small. Pascal was designed as a small, whole language with no plan to add to it at a later time. That was five-and-twenty years ago.

What came to pass?

Fortran has grown and grown. Many new words and new rules have been added. The new design is not bad; the parts fit well, one with the other. But to many programmers who have used Fortran for a long time, the Fortran of here and now is not at all the same as the language they first came to know and love. It looks strange.

PL/I has not grown much. It is, for the most part, just as it was when it first came out. It may be that this is just from lack of use. The flip side is that the lack of use may have two causes. Number one: PL/I was not designed to grow—it was designed to be all things to all who program right from the start. Number two: for its time, it started out large. No one knew all of PL/I; some said that no one *could* know all of PL/I.

Pascal grew just a tad and was used to build many large programs. One main fault of the first design was that strings were hard to use because they were all of fixed size. Pascal would have been of no use for the building of large programs for use in the real world if this had not been changed. But Wirth had not planned for the language to change in such ways, and in fact few changes were made.

At times we think of C as a small language designed from whole cloth. But it grew out of a smaller language called B, and has since grown to be a larger language called C plus plus. A language as large as C plus plus could not have spread so wide if it had been foisted on the world all at once. It would have been too hard to port.

(One more rule for making words: if we add the syllable "er" to a verb stem, we make a noun that names the person or thing that does what the verb says to do. For example, a buyer is one who buys. A user is one who does use.)

$$\langle \text{noun} \rangle ::= \langle \text{verb stem} \rangle \text{ “er”}$$

As you may by now have guessed, I am of like mind with my good friend Dick Gabriel, who wrote the well known screed "Worse Is Better."

```
http://www.ai.mit.edu/docs/articles/good-news/good-news.html
http://www.naggum.no/worse-is-better.html
```

(The real name was "Lisp: Good News, Bad News, How to Win Big," which is all words of just one syllable—which might seem like good luck for me, but the truth is that Dick Gabriel knew how to choose words with punch. Yet what first comes to mind for most persons is the part headed "Worse Is Better" and so that is how they cite it.) The gist of it is that the best way to get a language used by many persons is not to design and build "The Right Thing," because that will take too long. In a race, a small language with warts will beat a well designed language because users will not wait for the right thing; they will use the language that is quick and cheap, and put up with the warts. Once a small language fills a niche, it is hard to take its place.

Well, then, could not "The Right Thing" be a small language, not hard to port but with no warts?

I guess it could be done, but I, for one, am not that smart (or have not had that much luck). But in fact I think it can not be done. Five-and-twenty years ago, when users did not want that much from a programming language, one could try. Scheme was my best shot at it.

But users will not now with glad cries glom on to a language that gives them no more than what Scheme or Pascal gave them. They need to paint bits, lines, and boxes on the screen in hues bright and wild; they need to talk to printers and servers through the net; they need to load code on the fly; they need their programs to work with other code they don't trust; they need to run code in many threads and on many machines; they need to deal with text and sayings in all the world's languages. A small programming language just won't cut it.

So a small language can not do the job right and a large language takes too long to get off the ground. Are we doomed to use small languages with many warts because that is the sole kind of design that can make it in the world?

At one time this thought filled me with gloom. But then I saw a gap in my thinking. I said that users will not wait for "The Right Thing," but will use what comes first and put up with the warts. *But*—users will not put up with the warts for all time. It is not long till they scream and moan and beg for changes. The small language will grow. The warts will be shaved off or patched.

If one person does all the work, then growth will be slow. But if one lets the users help do the work, growth can be quick. If many persons work side by side, and the best work is added with care and good taste, a great deal can be added in a short time.

APL was designed by one man, a smart man—and I love APL—but it had a flaw that I think has all but killed it: there was no way for a user to grow the language in a smooth way. In most languages, a user can define at least some new words to stand for other pieces of code that can then be called, in such a way that the new words look like primitives. In this way the user can build a larger language to meet his needs. But in APL, new words defined by the user do not look like language primitives at all. The name of a piece of user code is a word, but things that are built in are named by strange glyphs. To add what look like new primitives, to keep the feel of the language, takes a real hacker and a ton of work. This has stopped users from helping to grow the language. APL has grown some, but the real work has been done by just the few programmers that have the source code. If a user adds to APL, and what he added seems good to the hackers in charge of the language, they might then make it be built in, but code to use it would not look the same; the user would have to change his code to a new form, because in APL a use of what is built in does not look at all like a call to user code.

Lisp was designed by one man, a smart man, and it works in a way that I think he did not plan for. In Lisp, new words defined by the user look like primitives and, what is more, all primitives look like words defined by the user! In other words, if a user has good taste defining new words, what comes out is a larger language that has no seams. The designer in charge can grow the language with close to no work on his part, just by choosing with care from the work of many users. And Lisp grew much faster than APL did, because many users could try things out and put their best code out there for other users to use and to add to the language. Lisp is not used quite as much as it used to be, but parts of it live on in other languages, the best of which is called *garbage collection* (and I will not try to tell you now what that means in words of one syllable—I leave it to you as a task to try in your spare time).

This leads me to claim that, from now on, a main goal in designing a language should be to plan for growth. The language must start small, and the language must grow as the set of users grows.

I now think that I, as a language designer helping out with the Java programming language, need to ask not "Should the Java programming language grow?" but "*How* should the Java programming language grow?"

There is more than one kind of growth and more than one way to do it. But, as we shall see, if the goal is to be quick and yet to do good work, one mode may be better by far than all other modes.

There are two kinds of growth in a language. One can change the vocabulary, or one can change the rules that say what a string of words means.

A *library* is a vocabulary designed to be added to a programming language to make the vocabulary of the programming language larger. *Libraries* means more than one library. A true library does not change the rules of meaning for the language; it just adds new words. (You can see from this that, in my view, the code that lets you do a "long jump" in C is not a true library.)

Of course, there must be a way for a user to make libraries. But the key point is that the new words defined by a library should look just like primitives of the language. Some languages are like this and some are not. Those that are not are harder to grow with the help of users.

It may be good as well to have a way to add to the rules of meaning for a language. Some ways to do this work better than other ways. But the language should let work done by a user look just like what was designed at the start. I would like to grow the Java programming language in such a way that users can do more of this.

In the same way, there are two ways to do the growing. One way is for one person (or a small group) to be in charge and to take in, test, judge, and add the work done by other persons. The other way is to just put all the source code out there, once things start to work, and let each person do as he wills. To have a person in charge can slow things down, but to have no one in charge makes it harder to add up the work of many persons.

The way that is faster and better than all other ways does both. Put the source code out there and let all persons play with it. Have a person in charge who is a quick judge of good work and who will take it in and shove it back out fast. You don't have to use what he ships, and you don't have to give back your work, but he gives all persons a fast way to spread new code to those who want it.

The best example of this way to do things is *Linux*, which is an *operating system*, which is a program that keeps track of other programs in a computer and gives each its due in space and time.

You ought to read what Eric Raymond had to say of how Linux came to be. I shall tell you a bit of it once I define two more words for you.

A *cathedral* is a huge church. It may be made of stone; it may fill you with awe; but the key thought, in the mind of Eric Raymond, is that there is but one design, one grand plan, which may take a long time—many years—to make real. As the years pass, few changes are made to the plan. Many, many persons are needed to build it, but there is just one designer.

A *bazaar* is a place with many small shops or stalls, where you can buy things from many persons who are there to sell their wares. The key thought here is that each one sells what he wants to sell and each one buys what he wants to buy. There is no one plan. Each seller or buyer may change his mind at whim.

Eric Raymond wrote a short work called "The Cathedral and the Bazaar" in which he looks at how programs are built or have been built in the past. (You can find it on his web site.)

`http://www.tuxedo.org/~esr/writings/cathedral-bazaar`

He talks of how he built a mail fetching program with the help of more than two hundred users. He quotes Fred Brooks as saying, "More users find more bugs," and backs him up with tales from this example of building a program in the bazaar style. As for the role of the programmer-in-charge, Eric Raymond says that it is fine to come up with good thoughts, but much better to know them when you see them in the work of other persons. You can get a lot more done that way. Linux rose to its heights of fame and wide use in much the same way, though on a much larger scale. (To take this thought to the far end of the line: it may be that one could write an operating system by putting a million apes to work at a million typing-machines, then just spotting the bits of good work that come out by chance and pasting them up to make a whole. But that might take a long time, I guess. Too bad!)

But the key point of the bazaar is not that you can get many persons to work with you at a task, for cathedral-builders had a great deal of help, too. Nor is the key point that you get help with the designing as well as with the building, though that in fact is a big win. No, the key point is that in the bazaar style of building a program or designing a language or what you will, the plan can change in real time to meet the needs of those who work on it and use it. This tends to make users stay with it as time goes by; they will take joy in working hard and helping out if they know that their wants and needs have some weight and their hard work can change the plan for the better.

Which brings me to the high point of my talk. It seems, in the last few years, at least, that if one is asked to speak at this meeting, one must quote Christopher Alexander. I know a bit of his work, though not a lot, and I must say thanks to Dick Gabriel for pointing out to me a quote that has a lot to do with the main point of this talk.

I am sad to say that I do not know what this quote means, because Christopher Alexander tends to use many words of more than one syllable and he does not define them first. But I have learned to say these words by rote and it may be that you out there can glean some thoughts of use to you.

Christopher Alexander says:

> Master plans have two additional unhealthy characteristics. To begin with, the existence of a master plan alienates the users ... After all, the very existence of a master plan means, by definition, that the members of the community can have little impact on the future shape of their community, because most of the important decisions have already been made. In a sense, under a master plan people are living with a frozen future, able to affect only relatively trivial details. When people lose the sense of responsibility for the environment they live in, and realize that they are merely cogs in someone else's machine, how can they feel any sense of identification with the community, or any sense of purpose there? (from *The Oregon Experiment*)

I think this means, in part, that it is good to give your users a chance to buy in and to pitch in. It is good for them and it is good for you. (In point of fact, a number of cathedrals were built in the bazaar mode.)

Does this mean, then, that it is of no use to design? Not at all. But in stead of designing a thing, you need to design a way of doing. And this way of doing must make some choices now but leave other choices to a later time.

Which brings me to the word you all want to hear: a *pattern* is a plan that has some number of parts and shows you how each part turns a face to the other parts, how each joins with the other parts or stands off, how each part does what it does and how the other parts aid it or drag it down, and how all the parts may be grasped as a whole and made to serve as one thing, for some higher goal or as part of a larger pattern. A pattern should give hints or clues as to when and where it is best put to use. What is more, some of the parts of a pattern may be holes, or slots, in which other things may be placed at a later time. A good pattern will say how changes can be made in the course of time. Thus some choices of the plan are built in as part of the pattern, and other choices wait till the time when the pattern is to be used. In this way a pattern stands for a design space in which you can choose, on the fly, your own path for growth and change.

It is good to design a thing, but it can be far better (and far harder) to design a pattern. Best of all is to know when to use a pattern.

Now for some more computer words.

A *datum* is a set of bits that has a meaning; *data* is the mass noun for a set of datums.

An *object* is a datum the meanings of whose parts are laid down by a set of language rules. In the Java programming language, these rules use types to make clear which parts of an object may cite other objects.

Objects may be grouped to form classes. Knowing the class of an object tells you most of what you need to know of how that object acts.

Objects may have fields; each field of an object can hold a datum. Which datum it holds may change from time to time. Each field may have a type, which tells you what data can be in that field at run time (and, what is more, it tells you what data can not be in that field at run time).

A *method* is a named piece of code that is part of an object. If you can name or cite an object, then you can call a method of that object; the method then does its thing.

The Java programming language has objects and classes and fields and methods and types. Now I shall speak of some things that the Java programming language does not yet have.

A *generic type* is a map from one or more types to a type. Put an other way, a generic type is a pattern for building types. A number of groups of persons have set forth ways to add generic types to the Java programming language; each way has its own good points. We heard three talks on adding generic types to the Java programming language at this meeting less than three-and-twenty hours ago.

An *operator* is a glyph, such as a plus sign, that can be used in a language as if it were a word. In C or the Java programming language, as in English, the sign first known as "and per se and" is an operator, but a full stop or quote mark is not an operator.

A word is said to be *overloaded* if it is made to mean two or more things and the hearer has to choose the meaning based on the rest of what is said. For example, using the rules defined near the start of this talk, a verb form such as "painted" might be a past tense or a past participle, and it is up to you, the hearer, to make the call as to which I mean when I say it.

At some times, by some persons, an overloaded word is called *polymorphic*, which means that the word has many forms; but the truth is that the word has but one form, and many meanings. The definition of a word may be polymorphic, but the word as such is not.

An operator can be overloaded in C plus plus, but right now operators in the Java programming language can not be overloaded by the programmer, though names of methods may be overloaded. I would like to change that.

I have said in the past, and will say now, that I think it would be a good thing for the Java programming language to add generic types and to let the user define overloaded operators. Just as a user can code methods that can be used in just the same way as methods that are built in, the user ought to have a way to define operators for user-defined classes that can be used in just the same way as operators that are built in. What is more, I would add a kind of class that is of light weight, one whose objects can be cloned at will with no harm and so could be kept on a stack for speed and not just in the heap. Classes of this kind would be well-suited for use as user-defined number types but would have other uses, too. You can find a plan for all this on a web page by James Gosling.

`http://www.javasoft.com/people/jag/FP.html`

(There are a few other things we could add as well, such as tail calls and ways to read and write those machine flags for numbers whose points float. But these are small language tweaks next to generic types and overloaded operators.)

If we grow the language in these few ways, then we will not need to grow it in a hundred other ways; the users can take on the rest of the task. To see why, think on these examples.

A *complex number* is a pair of numbers. There are rules for how to find the sum of two complex numbers, or a complex number times a complex number:

$$(a, b) + (c, d) = (a + c, b + d)$$

$$(a, b) \cdot (c, d) = (a \cdot c - b \cdot d, a \cdot d + b \cdot c)$$

Some programmers like to use them a lot; other programmers do not use them at all. So should we make "complex number" a type in the Java programming language? Some say yes, of course; other persons say no.

A *rational number* is a pair of numbers. There are rules (not the same as the rules for complex numbers, of course) for how to find the sum of two rational numbers, or a rational number times a rational number:

$$(a, b) + (c, d) = (a \cdot d + b \cdot c, b \cdot d)$$

$$(a, b) \cdot (c, d) = (a \cdot c, b \cdot d)$$

A few programmers like to use them a lot; most do not use them at all. So should we make "rational number" a type in the Java programming language?

An *interval* is a pair of numbers. There are rules (not the same as the rules for complex numbers or rational numbers, of course) for how to find the sum of two intervals, or an interval times an interval:

$$(a, b) + (c, d) = (a + c, b + d)$$

$$(a, b) \cdot (c, d) = (\min(a \cdot c, a \cdot d, b \cdot c, b \cdot d), \max(a \cdot c, a \cdot d, b \cdot c, b \cdot d))$$

A few programmers like to use them a lot and wish all the other programmers who use numbers would use them, too; but most do not use them at all. So should we make "interval" a type in the Java programming language?

John Horton Conway once defined a game to be a pair of sets of games (see his book *On Numbers and Games*), then pointed out that some games may be thought of as numbers that say how many moves it will take to win the game. There are rules for how to find the sum of two games, and so on:

$$\begin{aligned}
(A, B) + (C, D) &= \Big(\{\, a + (C, D) \mid a \in A \,\} \cup \{\, (A, B) + c \mid c \in C \,\}, \\
&\qquad \{\, b + (C, D) \mid b \in B \,\} \cup \{\, (A, B) + d \mid d \in D \,\}\Big)
\end{aligned}$$

$$-(A, B) = \Big(\{\, -b \mid b \in B \,\}, \{\, -a \mid a \in A \,\}\Big)$$

$$A - B = A + (-B)$$

$$\begin{aligned}
(A, B) \cdot (C, D) = \Big(&\{\, a \cdot (C, D) + (A, B) \cdot c - a \cdot c \mid a \in A, c \in C \,\} \cup \\
&\{\, b \cdot (C, D) + (A, B) \cdot d - b \cdot d \mid b \in B, d \in D \,\}, \\
&\{\, a \cdot (C, D) + (A, B) \cdot d - a \cdot d \mid a \in A, d \in D \,\} \cup \\
&\{\, b \cdot (C, D) + (A, B) \cdot c - b \cdot c \mid b \in B, c \in C \,\}\Big)
\end{aligned}$$

From this he worked out for hundreds of kinds of real games how to know which player will win. I think, oh, three persons in the world want to use this kind of number. Should we make it a type in the Java programming language?

A *vector* is a row of numbers all of the same type, with each place in the row named by the kind of number we first spoke of in this talk. There are rules ... In fact, for vectors of length three there are two ways to do "a vector times a vector," so you can have twice the fun!

$$(a, b, c) + (d, e, f) = (a + d, b + e, c + f)$$

$$(a, b, c) \cdot (d, e, f) = a \cdot d + b \cdot e + c \cdot f$$

$$(a, b, c) \times (d, e, f) = (b \cdot f - c \cdot e, c \cdot d - a \cdot f, a \cdot e - b \cdot d)$$

Vectors of length three or four are a great aid in making bits on the screen look like scenes in the real world. So should we make "vector" a type in the Java programming language?

A *matrix* is a set of numbers laid out in a square. And there are rules (not shown here!). So should we make "matrix" a type in the Java programming language?

And so on, and so on, and so on.

I might say "yes" to *each* one of these, but it is clear that I *must* say "no" to *all of them*! And so would James Gosling and Bill Joy. To add all these types to the Java programming language would be just too much. Some parts of the programming vocabulary are fit for all programmers to use, but other parts are just for their own niches. It would not be fair to weigh down all programmers with the need to have or to learn all the words for all niche uses. We should not make the Java programming language a cathedral, but a plain bazaar might be too loose. What we need is more like a shopping mall, where there are not quite as many choices but most of the goods are well designed and sellers stand up and back what they sell. (I could speak at length on the ways in which a shopping mall is like a cathedral—but not here, not now!)

Now, the user could define objects for such numbers to have methods that act in the right ways, but code to use such numbers would look strange. Programmers used to adding numbers with plus signs would kvetch. (In fact, the Java programming language does have a class of "big numbers" that have methods for adding them and telling which is larger and all the rest, and you can not use the plus sign to add such numbers, and programmers who have to use them do kvetch.)

Generic types and overloaded operators would let a user code up all of these, and in such a way that they would look in all ways just like types that are built in. They would let users grow the Java programming language in a smooth and clean way. And it would not be just for numbers; generic types would be good for coding hash sets, for example, and there are many other uses.

And each user would not have to code up such number classes, each for his own use. When a language gives you the right tools, such classes can be coded by a few and then put up as libraries for other users to use, or not, as they choose; but they don't have to be built in as part of the base language.

If you give a person a fish, he can eat for a day.

If you teach a person to fish, he can eat his whole life long.

If you give a person tools, he can make a fishing pole—and lots of other tools! He can build a machine to crank out fishing poles. In this way he can help other persons to catch fish.

Meta means that you step back from your own place. What you used to do is now what you see. What you were is now what you act on. Verbs turn to nouns. What you used to think of as a pattern is now treated as a thing to put in the slot of an other pattern. A meta-foo is a foo in whose slots you can put parts of a foo.

In a way, a language design of the old school is a pattern for programs. But now we need to "go meta." We should now think of a language design as being a pattern for language designs, a tool for making more tools of the same kind.

This is the nub of what I want to say. A language design can no longer be a thing. It must be a pattern—a pattern for growth—a pattern for growing the pattern for defining the patterns that programmers can use for their real work and their main goal.

My point is that a good programmer in these times does not just write programs. A good programmer builds a working vocabulary. In other words, a good programmer does language design, though not from scratch, but building on the frame of a base language.

In the course of giving this talk, because I started with a small language, I have had to define fifty or more new words or phrases and sixteen names of persons or things; and I laid out six rules for making new words from old ones. In this way I added to the base language. If I were to write a book, starting from just the words of one syllable, I am sure that I would have to define hundreds of new words. It should give no one pause to note that the writing of a program having a million lines of code might need many, many hundreds of new words—that is to say, a new language built up on the base language. I will be so bold as to say that it can be done in no other way. Well—there may be one other way, which is to use a large, rich programming language that has grown in the course of tens or hundreds of years, that has all we need to say what we want to say, that we are taught as we grow up and take for granted. It may be that a hundred years from now there will be a programming language that by then has stood the test of time, needs no more changes for most uses, and is used by all persons who write programs because each child learns it in school. But that is not where we are now.

So. Language design is not at all the same kind of work it was thirty years ago, or twenty years ago. Back then, you could set out to design a whole language and then build it by your own self, or with a small team, because it was small and because what you would then do with it was small.

Now programs are big messes with many needs. A small language won't do the job. If you design a big language all at once and then try to build it all at once, you will fail. You will end up late and some other language that is small will take your place.

It would be great if there were some small programming language that felt large, the way Basic English is small but feels large in some ways. But I don't know how to do it and I have good cause to doubt that it can be done at all. In its day, APL was a small language that felt large; but our needs have grown and APL did not have a good pattern for growth.

So I think the sole way to win is to plan for growth with help from users. This is a win for you because you have help. This is a win for the users because they get to have their say and get to bend the growth to their needs. But you need to have one or more persons, too, or one or more groups, to take on the task of judging and testing and sifting what the users do and say, and adding what they think best to the big pile of code, in the hope that other users will trust what they say and not have to go to all the work to test and judge and sift each new claim or each new piece of code, each for his own self.

Parts of the language must be designed to help the task of growth. A good set of types, ways for a user to define new types, to add new words and new rules to the language, to define and use all sorts of patterns—all these are needed. The designer should not, for example, define twenty kinds of number types in the language. But there will be users who, all told, beg for twenty kinds of numbers. The language should have a way for the user to define number types that work well with each other, and with plus signs and other such signs, and with the many ways of pushing bits in and out of the computer. One might define just one or two number types at the start, to show how it ought to be done. Then leave the rest to the users. Help them all as best you can to work side by side (and not nose to nose).

You may find that you need to add warts as part of the design, so that you can get it out the door fast, with the goal of taking out the warts at a later time. Now, there are warts and then there are warts! With care, one can design a wart so that it will not be too hard to take out or patch up later on. But if you do not take care at the start, you may be stuck for years to come with a wart you did not plan.

Some warts are not bad things you put in, but good things you leave out. Have a plan to add those good things at a later time, if you should choose to do so, and make sure that other parts of your design don't cut you off from adding those good things when the time is right.

I hope to bring these thoughts to bear on the Java programming language. The Java programming language has done as well as it has up to now because it started small. It was not hard to learn and it was not hard to port. It has grown quite a bit since then. If the design of the Java programming language as it is now had been put forth three years ago, it would have failed—of that I am sure. Programmers would have cried, "Too big! Too much hair! I can't deal with all that!" But in real life it has worked out fine because the users have grown with the language and learned it piece by piece, and they buy in to it because they have had some say in how to change the language.

And the Java programming language needs to grow yet some more—but, I hope, not a lot more. At least, I think only a few more rules are needed—the rest can be done with libraries, most of them built by users and not by Sun.

If we add hundreds of new things to the Java programming language, we will have a huge language, but it will take a long time to get there. But if we add just a few things—generic types, operator overloading, and user-defined types of light weight, for use as numbers and small vectors and such—that are designed to let users make and add things for their own use, I think we can go a long way, and much faster. We need to put tools for language growth in the hands of the users.

I hope that we can, in this way or some other way, design a programming language where we don't seem to spend most of our time talking and writing in words of just one syllable.

One of the good things I can say for short words is that they make for short talks. They gave me an hour and a half for this talk and I have used less than an hour. Once I am done, you will have time to go out in the hall, have a cup of tea, and talk with your friends.

But first, I would like to tell you what I have learned from the task of designing this talk. In choosing to give up the many long words that I have come to know since I was a child, words that have many fine shades of meaning, I made this task much harder than it needed to be. I hope that you have not found it too hard on your ears. But I found that sticking to this rule made me think. I had to take time to think through how to phrase each thought. And there was this choice for each new word: is it worth the work to define it, or should I just stick with the words I have? Should I do the work of defining a new word such as "mirror," or should I just say "looking glass" each time I want to speak of one? (As an example, I was tempted more than once to state the "ly" rule for making new words that change what verbs mean, but in the end I chose to cast all such words to one side and make do. And I came that close to defining the word "without," but each time, for better or for worse, I found some other way to phrase my thought.)

I learned in my youth, from the books of such great teachers of writing as Strunk and White, that it is better to choose short words when I can. I should not choose long, hard words just to make other persons think that I know a lot. I should try to make my thoughts clear; if they are clear and right, then other persons can judge my work as it ought to be judged.

From the work of planning this talk, in which I have tried to go with this rule much more far than in the past, I found that for the most part they were right. Short words work well, if I choose them well.

Thus I think that programming languages need to be more like the languages we speak—but it might be good, too, if we were to use the languages we speak more in the way that we now use programming languages.

All in all, I think it might be a good thing if those who rule our lives—those in high places who do the work of state, those who judge what we do, and most of all those who make the laws—were made to define their terms and to say all else that they say in words of one syllable. For I have found that this mode of speech makes it hard to hedge. It takes work, and great care, and some skill, to find just the right way to say what you want to say, but in the end you seem to have no choice but to talk straight. If you do not veer wide of the truth, you are forced to hit it dead on.

I urge you, too, to give it a try.