Philip Hood
hood@panix.com

"a triptych in fragments of modern perl."

dc.pm
2 October 2012
0. Modern, Post Modern
----------------------

Larry Wall speech,
Perl, the first postmodern computer language,
Linux World on Wednesday, March 3 1999
Perl, the first postmodern computer language

Thank you all for coming. I was hoping the title of this talk would scare away everyone who shouldn't be here.

Obviously I should have made up a scarier title...

Perl meets Godzilla?

Bambi meets Perl?

I just did an interview for Feed Magazine. When I read their intro to it, I was interested to see that they made reference to my "curious speeches." I'll take that as a compliment. Why I take it as a compliment is the subject of this speech. That's assuming this speech actually has a subject, which is still in doubt. Hey, at least it has a title. That's something.

By the way, I'm planning to leave some amount of time at the end for Q and A, so you should start thinking about the Q part while I'm talking.

When I was invited to talk here, it occurred to me that most of the people here would be more interested in Linux than in Perl, so, in the interests of universal harmonic convergence, I thought I should talk about both Perl and Linux. To do that, I had to figure out what Perl and Linux have in common. Besides the obvious, of course.

Obviously, both Perl and Linux owe a lot to Unix culture, but this is well documented. If I merely pointed out the obvious commonalities, I'd have to talk the whole time about things you can find out from the manuals. (Or should I say, things you ought to be able to find out from the manuals? Whatever.)

I'm not here today to teach you how to use Perl or Linux. I'm not here to teach you what Perl or Linux are.

I'm here to talk about why Perl and Linux have both been so successful. Note that I'm measuring success here not so much in terms of numbers of users, but in terms of satisfaction of users.

So I started thinking about deeper connections between Perl and Linux, and that led me to think more about the deeper reasons for writing software. And that led to the subject of this talk. I'm going to start off talking by about postmodernism. After that, I'll switch to talking about postmodernism. And at the conclusion, I'll return to the subject of postmodernism.

However, since this talk is itself a postmodern work of art, I'll be dragging in all sort of other cool things along the way, so maybe you won't fall asleep.

Nowadays people are actually somewhat jaded by the term "postmodern". Well, perhaps jaded is an understatement. Nauseated might be more like it. But, anyway, I still distinctly remember the first time I heard it back in the '70s. I think my jaw fell and bounced off the floor several times. To me it was utterly inconceivable that anything could follow modern. Isn't the very idea of "modern" always associated with the ideas "new" and "now"?

The idea was so inconceivable to me that it took me at least ten seconds to figure it out. Or to think I'd figured it out. As a musician, the pat answer occurred to me almost immediately. I was familiar with the periods of music: Baroque, Classical, Romantic, and Modern. Obviously, if there were to be a period of music following the Modern, it would have to be called something other than Modern. And postmodern is as good a name as any, especially since it's a bit of a joke on the ordinary meaning of modern. Obviously the Modern period was misnamed.

But, as I said, that was the pat answer. The Modern period was not misnamed. True, the ordinary word "modern" is associated with the "new" and "now", but the historical period we call Modern chose to associate itself with the "new" and the "now" in such a deep way that we actually see the breakdown of the whole notion of periods. The Modern period is the period that refuses to die. The world is now an odd mix of the Modern and the postmodern. Oddly, it's not just because the Modern refuses to die, but also because the postmodern refuses to kill the Modern. But then, the postmodern refuses to kill anything completely.
For example, it's been several decades now since a certain set of Bible translations came out, and you'll notice a pattern: the New English Bible, the New American Standard Bible, and the New International Version, to name a few. It's really funny. I suspect we'll still be calling them "new this" and "new that" a hundred years from now. Much like New College at Oxford. Do you know when New College was founded. Any guesses? New College was new in 1379.

A couple of days ago I was discussing all this with my daughter on the way to school. As usual, I turned on the radio to hear the news, and Heidi immediately started surfing all the music stations. Since this is one of the perils of fatherhood, I only said, "I have to talk about postmodernism on Wednesday. What should I say?"

She said, "Like, it's all about how you don't have justify everything with a reason anymore. You can just put in stuff because you like it, you know, because it's cool. With Modern stuff you always had to justify everything."

I said, "I still feel like I have to justify Perl all the time to a lot of people."

She settled on a station with some interesting music, and said, "This is Dave Matthews' Band. The thing that's really cool about him is that he, like, went out and found all these different artists who have different styles, and combined them all in ways you've never heard before."

I said, "Isn't it interesting how postmodernism has become so much a part of our culture that it's sort of fading into the woodwork?"

Heidi frowned and said, "Dude, dad, it's not like it's some kind of a fad. Postmodernism is deeper than that--it really is the culmination of everything that went before it. Like, it's all about coming full circle. It's not like we're going to stop wanting to do that next week."

I said, "I suspect you're right. After all, the various earlier periods of music were measured in centuries."

"It's not just music," she said.

"Well, of course not," I replied, "all these things go together, but some disciplines change at different rates. The reason I'm giving this talk on Wednesday is because I think there's still a big streak of Modernism running through the middle of computer science, and a lot of people are out of touch with their culture. On the other hand, I'm not really out to fight Modernism, since postmodernism includes Modernism as just another valid source of ideas. In fact, Perl contains lots of modern ideas from computer science. Along with all the rest of the ideas in there."

Heidi said, "You wanna know something really funny. In my IMP class, our class slogan is, 'There's more than one way to do it.'"

"You're kidding," I said. [I should also say that that IMP stands for Interactive Math Program, which is a math curriculum in which you sort of learn everything at once. In sort of a postmodern way.] Anyway, I said, "You're kidding."

"No," she said, "That's why IMP is better for math students like me--we learn better when we can see the big picture, and how everything fits in. The old way of learning math never gave you any context."

While I was digesting this, and thinking about how it applied to computer science, she went on, "Well, it's like, you know, we have this saying at school, when somebody gets uptight about something, we say: 'Tsall good. If someone is depressed, we say: 'Tsall good.'"

"But you don't actually think everything is good, do you?"

"No, of course not."

"Are you saying that everything has good elements in it?"

"No, Dad, I think when we say that, we're saying that, overall, things are good. Like, look at the big picture, don't just focus in on the two or three bad things that are happening to you right now."

I report this conversation to you not just because I think my kids are cute and smart, but also because I think it's important that we know where our culture is going, and because it's our kids that will shape our culture in the future. I don't think I could have defined postmodernism better than Heidi. Look at
the big picture. Don't focus in on two or three things to the exclusion of other things. Keep everything in context. Don't go out of your way to justify stuff that's obviously cool. Don't ridicule ideas merely because they're not the latest and greatest. Pick your own fashions. Don't let someone else tell you what you should like. 'Tsall good.

That's all well and good, but I ask you, if it's all good, why, in every other breath, does my daughter say "That sucks."?

There's a mystery here, and if we can fathom it, perhaps we'll learn a thing or two. I think that what's going on here is that our culture has undergone a basic shift, one that is actually healthy. It used to be that we evaluated everything and everyone based on reputation or position. And the basic underlying assumption was that we all had to agree whether something (or someone) was good or bad. Most of us actually used to believe in monoculturalism. Although even back then, we didn't really practice it. And in fact, you could argue that the whole point of Modernism was to break our cultural assumptions. We could argue all day long about whether postmodernism came about because Modernism succeeded or because it failed. To some extent.

This would bother a Modernist, because a Modernist has to decide whether this is true OR that is true. The Modernist believes in OR more than AND. Postmodernists believe in AND more than OR. In the very postmodern Stephen Sondheim musical, _Into the Woods_, one of the heroines laments, "Is it always or, and never and?" Of course, at the time, she was trying to rationalize an adulterous relationship, so perhaps we'd better drop that example. Well, hey. At least we can use Perl as an example. In Perl, AND has higher precedence than OR does. That proves Perl is a postmodern language.

But back to the monoculturism of Modernism, or rather the assumption of monoculturalism. Nowadays we've managed to liberate ourselves from that assumption, by and large (where by and large doesn't yet include parts of the Midwest). This has had the result that we're actually free to evaluate things on the basis of what's actually good and what's actually bad, rather than having to take someone's word for it.

More than that, we're required to make individual choices, the assumption being that not everyone is going to agree, and that not everyone should be required to agree. However, in trade for losing our monoculturalism, we are now required to discuss things. We're not required to agree about everything, but we are required to at least agree to disagree. Since we're required to discuss things, this has the effect that we tend to "deconstruct" the things we evaluate. I'll talk more about the pros and cons of deconstructionism in a bit, but let me just throw out an example to wake you up.

The most deconstructed man on the planet right now is Bill Clinton. The public, and later the Senate, chose to evaluate Bill Clinton's morality separately from Clinton's fitness to govern. I'm not going to comment on whether I agree with that decision, but I'd just like to point out that this could not have happened thirty or forty years ago. We were not postmoderns back then. We had to have a whole president, or no president, so people conspired to make sure we kept a whole president (even though there was probably just as much hanky panky going on back then as there is now). Everything used to be in black and white, like our TVs. We kept our presidents looking good until we got one we couldn't make look good, and then everyone switched to making the president look bad for a while. But we never did deconstruct Nixon the way we've deconstructed all the presidents since Nixon. Nixon is still monolithic, even though we've managed to bypass him and deconstruct Kennedy in hindsight. Perhaps it's all related to the saying that "Beauty is skin deep, but ugly goes right to the bone."

If that's the case, I pity the person who's only skin and bones.

Now you may be wondering what all this has to do with Perl. So am I. I'll think of something presently.

Time passes...

While I'm thinking about the next thing to say in my talk, let me say a bit more about deconstructionism. I do not view deconstructionism as a form of postmodernism so much as I view deconstructionism as the bridge between Modernism and postmodernism. Modernism, as a form of Classicalism, was always striving for simplicity, and was therefore essentially reductionistic. That is, it tended to take things to pieces. That actually hasn't changed much. It's just that Modernism tended to take one of the pieces in isolation and glorify it, while postmodernism tries to show you all the pieces at once, and how they relate to each other.

For instance, this talk. If this were a Modern talk, I'd try to have one major point, and drive it into the ground with many arguments, all coherently arranged. Instead, however, I let you see that there's a
progression in my own thought process as I'm writing. I would pause in my talk at the same point that I paused in my thought process. If I were a journalist, I'd spend as much time talking about my angst in covering the story as I'd spend covering the actual story. And if I were building a building instead of writing a talk, I'd let the girders and ductwork show. These are all forms of deconstructionism.

I'm still trying to think about how this relates to Perl, by the way.

More time passes...

I first heard about postmodernism in the late '70s at Seattle Pacific University from my wife's Lit Crit professor, Dr. Janet Blumberg. Postmodernism came early to literature, so it's no surprise that we heard it first from a literary critic. By the way, don't think of literary critics like you think of theatre critics. Literary critics usually know what they're talking about.

Even if they're wrong.

Anyway, we heard it first from Dr. Blumberg, who was never wrong, so naturally we first thought about it in terms of literature. In fact, most people still think of postmodernism as a kind of weird literature. But postmodernism was also coming along in architecture too, as we were soon to find out. Seattle Pacific was wanting to build a new science and math building, so they decided to recycle an old warehouse down by the ship canal. Note the first element of postmodernism there--they were reusing something old, taking the good parts, and leaving behind the bad parts, though they probably didn't say to themselves, "This rules," or "That sucks." But I'm sure they thought it. Anyway, they combined the old with modern ideas about having a large open lab inside, and making the whole building solar heated. They made it a comfy place at the same time, with a sunken study area containing sofas. And they made all the girders and ductwork show, because they thought it was cool. They also did it because it was postmodern, though they didn't know that yet.

I think I know now how this relates to Perl.

When I started writing Perl, I'd actually been steeped in enough postmodernism to know that that's what I wanted to do. Or rather, that I wanted to do something that would turn out to be postmodern, because you can't actually do something postmodern, you can only really do something cool that turns out to be postmodern. Hmm. Do I really believe that? I dunno. Maybe. Sometimes. You may actually find this difficult to believe, but I didn't actually set out to write a postmodern talk. I was just going to talk about how Perl is postmodern. But it just kind of happened. So you get to see all the ductwork.

Anyway, back to Perl. When I started designing Perl, I explicitly set out to deconstruct all the computer languages I knew and recombine or reconstruct them in a different way, because there were many things I liked about other languages, and many things I disliked. I lovingly reused features from many languages. (I suppose a Modernist would say I stole the features, since Modernists are hung up about originality.) Whatever the verb you choose, I've done it over the course of the years from C, sh, csh, grep, sed, awk, Fortran, COBOL, PL/I, BASIC-PLUS, SNOBOL, Lisp, Ada, C++, and Python. To name a few. To the extent that Perl rules rather than sucks, it's because the various features of these languages ruled rather than sucked.

But note something important here. I left behind more than I took. A lot more. In modern terms, there was a lot of stuff that sucked. Now, on the feature set issue, Perl is always getting a lot of bad press.

I think people who give bad press to Perl's feature set should have more angst about their reporting.

I picked the feature set of Perl because I thought they were cool features. I left the other ones behind because I thought they sucked.

More than that, I combined these cool features in a way that makes sense to me as a postmodern linguist, not in a way that makes sense to the typical Modernistic computer scientist. Recall that the essence of Modernism is to take one cool idea and drive it into the ground. It's not difficult to look at computer languages and see which ones are trying to be modern by driving something into the ground. Think about Lisp, and parentheses. Think about Forth, and stack code. Think about Prolog, and backtracking. Think about Smalltalk, and objects. (Or if you don't want to think about Smalltalk, think about Java, and objects.)

Think about Python, and whitespace. Hi, Guido.

Or think about shell programming, and reductionism. How many times have we heard the mantra that a program should do one thing and do it well?
Well...Perl does one thing, and does it well. What it does well is to integrate all its features into one language. More importantly, it does this without making them all look like each other. Ducts shouldn't look like girders, and girders shouldn't look like ducts. Neither of those should look like water pipes, and it's really important that water pipes not look like sewer pipes. Or smell like sewer pipes. Modernism says that we should make all these things look the same (and preferably invisible). Postmodernism says it's okay for them to stick out, and to look different, because a duct ought to look like a duct, and a sewer pipe ought to look like a sewer pipe, and hammer ought to look like a hammer, and a telephone ought to look like either a telephone, or a Star Trek communicator. Things that are different should look different.

You've all heard the saying: If all you have is a hammer, everything starts to look like a nail. That's actually a Modernistic saying. The postmodern version is: If all you have is duct tape, everything starts to look like a duct. Right. When's the last time you used duct tape on a duct?

The funny thing is, Modernism itself was a kind of hammer, and it made everything look like something to be hammered. The protest movement of the '60s was Modernistic: "If I had a hammer, I'd hammer all over this land." The focus was always on the nail, or on whatever it was that was getting pounded. And many things did get hammered in the Modern age. Architectural beauty, for one. That one is obvious just by looking at the skyline of any major city. It's easy to tell which buildings were built in the 50's and 60's. They're the ones that look like boxes. When we first saw them, we thought they looked very modern. Well, they did. But when the Seattle First National Bank was built in, you guessed it, Seattle, we all made jokes about how it looked like the box the Space Needle came in. At least the Space Needle was cute, kinda like the Jetsons were cute. But the Space Needle wasn't really very functional, unless you go in for rotating restaurants.

In fact, at many different levels, Modernism brought us various kinds of dysfunction. Every cultural institution took a beating. Government took a beating. Schools took a beating. Certainly the family took a beating. Everyone took a beating, because Modernism was about attacking problems. Modernism was the hammer. (I'd like to make a pun on hammer and sickle here, but I'm not sure what it would be. Certainly Russia was more hammered than we were by Modernism, in the cloak of Marxism. I know what it means to be hammered, but I'm still trying to figure out what it would mean to be more sickled. Hmm. Unless that's talking about the Grim Reaper. Russia has a lot of experience with that too.) Anyway, back to our talk. Modernism oversimplifies. Modernism puts the focus squarely on the hammer and the nail.

In contrast, postmodernism puts the focus back onto the carpenter. You'll note that carpenters are allowed to choose whether or not to use hammers. They can use saws and tape measures if they choose, too. They have some amount of free will in the matter. They're allowed to be creative. Especially if they're working on Mrs. Winchester's house. Hey, it's right down the road, if you don't believe me.

So, to drag the subject back to computers, one of the characteristics of a postmodern computer language is that it puts the focus not so much onto the problem to be solved, but rather onto the person trying to solve the problem. I claim that Perl does that, and I also claim that, on some level or other, it was the first computer language to do that. I'd also like to claim that, in many ways, it's still the only language to do that.

How does Perl put the focus onto the creativity of the programmer? Very simple. Perl is humble. It doesn't try to tell the programmer how to program. It lets the programmer decide what rules today, and what sucks. It doesn't have any theoretical axes to grind. And where it has theoretical axes, it doesn't grind them. Perl doesn't have any agenda at all, other than to be maximally useful to the maximal number of people. To be the duct tape of the Internet, and of everything else. You've heard the joke, I'm sure. How is duct tape like the Force? It has a light side, and a dark side, and it holds the universe together. Later in this talk, I intend to define the universe and give three examples.

I have to be honest here. I'm with Linus--I personally want to take over the world. I want to take over the world because I'm an egomaniac. A nice sort of egomaniac, an egomaniac moderated by belief in the value of humility, but an egomaniac nonetheless.

Fortunately, I am not Perl. Perl was my servant before it was anyone else's, so I taught Perl to be a better servant than I could ever teach myself to be. Perl is like the perfect butler. Whatever you ask Perl to do, it says "Very good, sir," or "Very good, madam." Only occasionally does Perl give you a stiff upper lip, or say "Tsk, tsk." But if you ask Perl its opinion, it will advise you on matters of taste. "I'm sorry sir, but bareword 'foo' is not allowed while 'strict subs' is in use."
Contrast that with the Modern idea of how a computer should behave. It's really rather patronizing: "I'm sorry Dave. I can't allow you to do that."

The trouble with having a submissive servant is that it puts the burden back on you to make the decisions. Come to think of it, that's the problem with having a submissive wife too. My wife is very submissive. She's always saying, "I submit this problem to you because I don't want to decide it." Ah, well.

If the burden of decision making is on the programmer, then it's possible for the programmer to make a mess of things. It's possible for Perl programmers to write messy programs. (In case you hadn't noticed.) It's also possible for Perl programmers to write extremely clean, concise, and beautiful programs.

Let me state my beliefs about this in the strongest possible way. The very fact that it's possible to write messy programs in Perl is also what makes it possible to write programs that are cleaner in Perl than they could ever be in a language that attempts to enforce cleanliness. The potential for greater good goes right along with the potential for greater evil. A little baby has little potential for good or evil, at least in the short term. A President of the United States has tremendous potential for both good and evil.

I do not believe it is wrong to aspire to greatness, if greatness is properly defined. Greatness does not imply goodness. The President is not intrinsically "gooder" than a baby. He merely has more options for exercising creativity, for good or for ill.

True greatness is measured by how much freedom you give to others, not by how much you can coerce others to do what you want. I remember praying a prayer when I was very young, not much more than a baby myself. "God is great. God is good. Let us thank him for our food. Amen." Well, I'm here to say amen to that. God's greatness and goodness are measured by the fact that he gives us choices. He doesn't require us to thank him for our food. (In case you hadn't noticed.) God is not a Modernist. He doesn't view us as nails. God expects us to behave like carpenters. Indeed, he gave us a carpenter as an example.

So I think God is postmodern. He has his own ideas of what rules, and what sucks, and he doesn't expect everyone else to agree with him. Mind you, he likes it when people agree with him. I like it when people agree with me about Perl. But I don't expect everyone to agree with me. Of course, some of my loyal followers expect everyone to agree with me. I try to think of it as an endearing characteristic. Personally, I think the Perl slogan, There's More Than One Way To Do It, applies outside of Perl as well as inside. I explicitly give people the freedom not to use Perl, just as God gives people the freedom to go to the devil if they so choose.

As long as we're in a demonizing frame of mind, please allow me to demonize Modernism a little more. True, postmodernism admits Modernism as one source of inspiration, along with Romanticism, Classicalism, and, er, uh, Baroqueism. Baroqueness? I always think it's a compliment when someone says Perl is baroque. I just think of J.S. Bach. He wrote a lot of way cool stuff. Handel also had his moments.

Anyway, back to Modernism. Postmodernism does draw some inspiration from Modernism. And, in fact, postmodernism could not have come about without Modernism before it, because deconstructionism is simultaneously Modern and postmodern, being both reductionistic and holistic. Be that as it may, Postmodernism has deconstructed Modernism and determined that large parts of it suck. In religious terms, Modernism can be viewed as a series of cults. And postmodernism is defined as an escape from those cults. A kind of deprogramming, if you will. Perhaps the title of this talk should have been, "Perl, the first postmodern DEprogramming language".

We won't go into the fact that "Perl culture" sometimes gets shortened to "Perl cult".

I have to give credit where credit is due here. And to show my ductwork. I didn't think of all this myself. I was flying up to Seattle with my wife and my daughter (yes, that one) because my daughter is thinking about attending Seattle Pacific University, that hotbed of subversive postmodernism. Surprise, surprise. So I asked my wife about the differences between Modernism and postmodernism. After all, one has to talk about something with one's spouse.

Especially in front of one's daughter.

Before I get into the list of Modernistic cults, though, I just remembered another cute story about Seattle Pacific. The school had commissioned a Modern Artist to produce a Modern Art, you know the
kind, a sculpture, if you can call it that, to be placed on the lawn out in front of student union building, on the corner of campus where anyone driving by could see it. It was most definitely Modern. It consisted of two large black surfaces, partly rounded and partly square, leaning against each other. It was actually rather hideous. You know the sort.

Well, one day we noticed that the large sculpture had had babies. There were seven or eight of the cute little beggars, perfect little replicas huddling around their mommy. It was wonderful. It was precious. It was funny. At least, it was funny until the Modern Artist came storming in and, with no sense of humor at all, removed his work of art, threatening never to have anything to do with Seattle Pacific again. Good riddance, we thought. And smiled. We’re still smiling. In case you hadn’t noticed.

Anyway, back to cults. The story I just told is illustrative of several of them. First of all, we have the Cult of Spareness. The example of Modern Art I just mentioned was very spare. It was minimalistic. It was almost an artless Art. Certainly the emotion it was trying to instill was something akin to hammering. We felt like nails.

Many modern computer languages aspire to be minimalistic. They either succeed in being minimalistic, in which case they’re relatively useless, or they don’t succeed in being truly minimalistic, in which case you can actually solve real problems with them. A number of languages give lip service to the idea of minimalism, but merely sweep the complexity of the problem under the carpet of the programmer. C is a minimalistic language, but only if you don’t count all the libraries that are necessary to use it usefully. C++ is obviously not trying to be minimalistic. Unix is considered by some to be a minimalistic operating system, but the fact of the matter is that if you think of Unix as a programming language, it’s far richer than even Perl. Perl is, by and large, a digested and simplified version of Unix. Perl is the Cliff Notes of Unix.

Unix (and by extension Linux) are really simultaneously Modern and postmodern. Unix philosophy is supposedly reductionistic, and minimalistic. But instead of being Modernistic, Unix is actually deconstructionistic. The saving grace of deconstructionism is that it is also reconstructionism. When you’ve broken everything down into bits, you’re required to put them back together again in a different way. In order to solve real problems, Unix requires you not only to be reductionistic, but also holistic. It’s no accident that the ductwork shows in shell scripts. Only we call them pipes.

Postmodernism isn’t afraid of ornamentation, because postmodernism is a retreat from classicalism back to romanticism. That particular pendulum is quite periodic. The Classical and Modern periods of art identified beauty with simplicity. The Baroque and Romantic periods of art identified beauty with complexity. I think it’s an interesting synchronicity that, even as our art is becoming more complex again, science is also discovering beauty in complexity theory. Perhaps it’s more than a synchronicity. Just as Modern art had exhausted the possibilities of bigger hammers, so had science. In short, we’d been oversimplifying for too long, and hence couldn’t see the simplicity within the complexity of a leaf. I would like to tell you that Perl is simple in its complexity. But some people won’t understand that. So pretend I didn’t say that, unless you do.

I could go on about simplicity, but let’s move on to the next cult. Modernism is also a Cult of Originality. It didn’t matter if the sculpture was hideous, as long as it was original. It didn’t matter if there was no music in the music. Plagiarism was the greatest sin. To have your work labeled "pastiche" was the worst insult. The only artistic endeavor in the Modern period not to suffer greatly from the Cult of Originality was architecture. Architecture went in for simplicity and functionalism instead. With the notable exception of certain buildings that were meant to look like Modern art, usually because they contained Modern art. Odd how that happens.

The Cult of Originality shows up in computer science as well. For some reason, many languages that came out of academia suffer from this. Everything is reinvented from first principles (or in some cases, zeroth principles), and nothing in the language resembles anything in any other language you’ve ever seen. And then the language designer wonders why the language never catches on.

No computer language is an island, either.

In case you hadn’t noticed, Perl is not big on originality. Come to think of it, neither is Linux. Does this bother you? Good, perhaps our culture really is getting to be more postmodern.

The next cult on the hit parade is the Cult of Seriousness. Recall how seriously our Modern Artist took himself and his art. He was unable to laugh at himself.

I find that there are a certain number of humorless people who don't appreciate, um, humor. There is a small but steady drizzle of messages into O'Reilly & Associates from people who are offended by the
fact that my book, the Camel book, contains jokes. It's really quite funny reading the messages from these people. It reminds me of the time I was sitting in a theatre in Palo Alto. It was intermission, and we were in the middle of watching a hilarious play about a dysfunctional family. You know, kind of like a sitcom, only done right. The audience just roared throughout the first act. Anyway, at intermission, this older guy with a deep, gruff voice behind me says to his wife, in all seriousness, "I don't see what's so funny. It's just like an ordinary family."

I confess, I had to cover my mouth with my hands to keep from guffawing like Tom Christiansen. That's got to be one of the funniest things I've ever heard. Well, maybe you had to be there.

Postmodernism is not afraid to laugh at itself. It's not afraid of cute, and it's not afraid of funky, and it's not afraid of what a Modernist would call kitsch. You know, it's actually kind of liberating to be going down the road, and be able to yell, "New buggie! Pea soup green." Postmoderns aren't afraid to be nostalgic about old slug bugs, either. Sentimentality is cool, if you're into that sort of thing. Retro rules. Unless it rocks. I don't know if sentimentality rules or rocks, but it's definitely cool.

As Heidi would say, "Dude, I'm stoked."

You'll notice I keep talking about my wife and my daughter. In case you hadn't noticed. The Modernist would of course explain to you that I was resorting to cheap sentimental tricks to try to establish an emotional bond with my audience. A postmodernist would, of course, agree. But the postmodernist will point out that cheap tricks are less expensive than costly tricks. Showing your ductwork is usually cheaper than hiding it. Even if it's not cheaper, it's certainly more entertaining. I certainly find the Iron Chef entertaining. That's definitely a postmodern show. There's lots of cool stuff on TV these days. My daughter enjoys Daria, because it's so cynical about everything, including itself. And if you'd like to see an unashamedly postmodern anime, you should get ahold of Revolutionary Girl Utena.

Well, enough of that. Let's see what's next in our talk. Oh, oh, here comes a biggie. The Cult of Objectivity.

You know, Modernism tried. It tried real hard. It really, really tried. It tried to get rid of conventions. It thought it got rid of conventions. But all it really did was make its conventions invisible. At least to itself.

Reductionists often feel like they're being objective. But the problem with reductionism is that, once you've split your universe into enough pieces, you can't keep track of them any more. Psychologists tell us that the human mind can only keep track of about seven objects, plus or minus two. That's for short-term memory. It gets both worse and better for long-term memory, but the principle still stands. If you lose track of something, it's because you thought it was less important, and didn't think about it often enough to remind yourself. This is what happened to Modernists in literature. They've forgotten what's important about literature.

Note how we still periodically hear the phrase "serious literature". This is literature that is supposedly about Real Life. Let me tell you something. The most serious literature I've ever read is by Lois McMaster Bujold. Any of you read her? It's also the funniest literature I've ever read. It's also space opera. "Genre fiction," sneers the Modernist. Meaning it follows certain conventions. So what? Nobody in the world can mix gravity and levity the way Bujold does in her Vorkosigan books. It's oh so definitely about real life. So what if it follows space opera conventions. Sonnets follow certain conventions too, but I don't see them getting sneered at much these days. Certainly they were always called "serious".

How long till Bujold becomes required reading in high school? Far too long, in my opinion. Horrors. We wouldn't want our students actually enjoying what they read. It's not--it's not Real Life.

As if the Lord of the Flies is real life. Feh.

Perl programming is unabashedly genre programming. It has conventions. It has culture. Perl was the first computer language whose culture was designed for diversity right along with the language. We're not objective about Perl, but as postmodernists, we freely admit that we're not objective, and we try to compensate for it when we want people to think we're objective. Or when we want to think ourselves objective. Or, at least, not objectionable.

Or, at least, not object-oriented.

I would like to say one thing here about objectivity, however. While I despise the Modern Cult of Objectivity, I also despise the quasi-postmodern Cult of Subjectivity. I call it absolute cultural
relativism. It's the notion that everything is as good as everything else, because goodness is only a matter of opinion. It's like claiming that the only thing you can know absolutely is that you can't know anything absolutely. I think this is really just another form of Modernism, a kind of existentialism really, though unfortunately it's come to be associated with postmodernism. But I think it sucks.

The funny thing is, it's almost right. It's very close to what I do, in fact, believe. I'd go so far as to call myself a strong postmodernist. Strong postmodernism says that all truth is created. But this really isn't a problem for anyone who believes in a Creator. All truths are created relative, but some are more relative than others. A universal truth only has to be true about our particular universe, so to speak. It doesn't much matter whether the universe itself is true or false, just as long as it makes a good story. And I think our universe does make a good story. I happen to like the Author.

I like Lois McMaster Bujold too, so I read her stories. Same for Tolkien, and C.S. Lewis. Turning that around, some people use Perl because they like me. Who am I to argue with them? You're all totally objective about Linus and Linux, right? Uh, huh. Three cheers for objectivity.

I'm getting tired of talking about cults, and you're probably getting tired of listening to me talk about cults. However, I want to talk about the open source phenomenon now, and I'm afraid I'll have to drag the cults in occasionally. But fear not. I think the open source movement is, actually, a postmodern movement.

Think about it. We've actually been doing open source for a couple of decades now. Why is it suddenly taking off now? Why not twenty years ago. Linux could have been written twenty years ago, albeit not by Linus.

Of course there are lots of mundane reasons why Linux wasn't written twenty years ago, not the least of which is that we didn't really have the ubiquitous, cheap hardware to support it yet. Nor did we have the networking to support cooperative development. But since this is a philosophical talk, I'll ignore reality and talk about what I think was really going on. Here's where the cults come back in again.

The Cult of Spareness decreed that we should all use the same operating system. Of course, everyone had their own idea of what that was, but Bill Gates actually had the most success in carrying out the decree. For which he is now on trial, where he may eventually have to consent to a consent decree. All in all, it's been a bad year to be named Bill. The wolves are circling, and waiting for further signs of weakness, and everyone's hedging their bets by attending LinuxWorld, and making sure the press know it.

Meanwhile, back on the Unix side of the universe, The Cult of Originality decreed that, if you were going to work on something, it had to be something new. Reimplementing an open source Unix would have been laughable, especially since any university could get the sources to Unix anyway for next to nothing. When other companies besides Ma Bell figured out that they should have an implementation of Unix, they all had to make it different enough that they could distinguish themselves in the marketplace. That is, what they wrote had to be original.

The Cult of Seriousness decreed that everyone had to keep their source code hidden, because other people might laugh at how bad our code is, and make us fix it. Or worse, someone might steal our bad code and make it better. Then we would be out of business, and Life is Serious Business. A peek at the source code for Unix is obviously worth $100,000, because we can get that much for it. Programming for the fun of it? Get real!

Finally, the Cult of Objectivity decreed that the way we always did business was obviously the only way to do business. Our culture is the only possible culture. There are no social conventions here. These aren't the droids you're looking for. Move along.

In short, think about what it takes to put together an open source project such as Linux or Perl. You need a lot of people who think programming is serious fun. You need a culture of sharing, which is just the flip side of a culture in which you can borrow things without shame. You need people who have been hammered into dysfunctionality long enough that they're looking for new ways to form communities. You need people who are willing to be partisan on behalf of their chosen culture, while remaining sufficiently non-partisan to keep in touch with the rest of the world. It's no fun to create a new culture and then cut it off from the rest of humanity. No, the fun thing is to try to persuade others to share your opinions about what rules and what sucks. Nothing is more fun than evangelism.

There are two kinds of joiners in the world. Think of it in terms of anthropology. There are the kinds of people who join a tribe, and kind of get sucked in, like a black hole. That's the last you hear from
them, unless you happen to be in the black hole with them. And we need people like this in our tribes, if only to be cheerleaders.

But the open source movement is energized by the other sort of joiner. This sort of person joins many tribes. These are the people who inhabit the intersections of the Venn diagrams. They believe in ANDs rather than ORs. They’re a member of more than one subset, more than one tribe. The reason these people are important is, just like merchants who go between real tribes, they carry ideas from one intellectual tribe to another. I call these people "glue people", because they not only join themselves to a tribe, they join tribes together. Twenty years ago, you couldn't easily be a glue person, because our culture was not yet sufficiently accepting of diversity. It was also not accepting of information sharing. If you got sucked in by Bell Labs, you might get out to the occasional Usenix, but that was about it. If you got sucked in by the NSA, nobody ever heard from you again. Come to think of it, that's still true.

Still and all, things have improved greatly, and the bridges across the gaps have gotten sturdier. Now people can send their memes across a wider chasms without getting crucified on one end of the bridge or the other. And as we started sending these memes across the chasms, what we discovered was that we didn't have a bunch of separate open source movements, but rather a single big open source movement. To be sure, it's a fuzzy, postmodern sort of movement, with lots of diversity, and a certain amount of turmoil, but it's about as good as any movement gets these days. We all suck at slightly different things, but we're in basic agreement that the old way of business sucked a lot worse that whatever it is we're doing now. We've agreed to agree. Except when we don't.

That sounds like it ought to be the end of my talk, but I still have a bunch of things to say, so I'll just keep going. Who knows, maybe it'll relate.

The other day, I was talking to a glue person whose name is Sharon Hopkins. Among other things, she's known as the Perl Poet, because she's written more poetry in Perl than anyone else. She also writes a kind of non-Perl poetry that was dubbed by another poet as "sharonesque". Here's a cute example:

I'd travel to the ends of time
For you, my one, my only love.
I'd force the sun to leave its track
(If you were lost) to fetch you back.
I'd suck the juices from a lime,
I'd re-write Moby Dick in rhyme,
I'd happily commit a crime!
For you, my dearest darling dove.
I'd do it all, and more beside --

Now *would* you take the trash outside?

Sharon Hopkins
Winter, 1989-90

I had to write a response to that poem. Actually, two responses. I won't inflict the longer one on you, but here's the shorter one:

I've taken the trash out innumerable times,
I've taken the trash out in inclement climes,
I've taken the trash out 'cuz that's what I do,
But I *won't* take the trash out when you tell me to.

Well, anyway, most of Sharon's poetry is relational, as befits a postmodern glue person. As I was saying, we were talking recently. Oddly, the subject was postmodernism. Fancy that. We were discussing how postmodern stuff can have goofy things mixed in with things that matter. She likes Iron Chef, too. That's a Japanese show where you have some seriously good cooking mixed with an extremely silly race to cook the best meal. Watching the judges judge the meals is the best part. We were also talking about Revolutionary Girl Utena, where we have the Absolute Destiny Apocalypse mixed in with octopi falling out of closets. Both shows are full of arbitrary but endearing conventions. Anyway, I said something about what I was going to talk about here, and she said an interesting thing: Yes, Modernism created a lot of dysfunction--nobody disputes that. We were encouraged to revolt, deconstruct, cut apart our papers, run away from home and take drugs, not get married, and so on. Modernism tore a lot of things apart, but especially the family. The
interesting thing to me is that postmodernism is propagating the dysfunction, because it actually finds its meaning in dysfunction. Postmodernism really is a result of Modernism.

For one thing, notice how you can't rebel by being dysfunctional any more. It's no longer interesting--we've done that already.

But it's more than that. Think of Perl culture as a dysfunctional family. Or think of the various communities that arise on the net. Think of our Gen X group at church and their obviously postmodern tastes: night club decor mixed with candles. But it's really about being together. Nowadays, family is where you find it. Family is where you create it.

I think Sharon knows what she's talking about here. She actually met her current boyfriend online, but don't tell her I said that. Anyway, I thought she has an interesting perspective on the way the net works nowadays. Imagine, open source is merely a byproduct of our need for family. So, look at all of you out there. You're just a big, dysfunctional family trying to create meaning. Don't look so nervous. I'm not going to call for a group hug.

Let's see, what else can I talk about. Did you realize how many things can be abbreviated "pm"? Prime Minister. Post Meridian. Post Modern. Perl Module. Perl Monger. Are there any Perl Mongers out there today? There you have it--yet another dysfunctional family. You guys can go out and have a group hug later in the pub.

Okay, let's see. "pm" is an abbreviation for Perl Module, which is why, of course, we use ".pm" is the extension for a Perl module. It used to be that we used ".pl" for Perl code. People still do use ".pl" in Windows, but that's because they're all still stuck back in the Modern age. Anyway, there's a funny thing about using ".pl" for the Perl extension. People used to argue a lot about what the next language after C would be. Everyone knew that the previous language had been called "B", after the first letter in BCPL, which came even earlier than B. The two proposed candidates for the next language were "D", because that's what comes after "C" in the alphabet, and "P", because that was the next letter in BCPL.

Well, as you can see, it didn't work out either way. One of the successors to C was C++, which is a cute pun on the autoincrement operator, but makes it an absolute pain to try to figure out what the proper extension for a C++ file should be. .C? .cpp? .cxx? I suppose as a postmodern person, I shouldn't mind the diversity, but somehow I do. I suppose a little inconsistency is good for the soul.

Anyway, the other successor to C gobbled up two letters instead of one. Which is why many Perl scripts have the extension, ".pl", finishing off BCPL. It's a pity, in a way. Now there can never be a language named "L". Perhaps it's just as well. Quite apart from the annual yuletide puns we'd get on "noel, noel", there's also the problem that people would have confused the language with lex, which already uses a ".l" extension. Since lex had already taken it, noel was available. So to speak. Sorry.


Or as Tiny Tim says, God bless us, every one.

If you guys want me to stop talking, you'd better ask some questions.
Jean Michel Basquiat, Untitled
since feeling is first

e.e. cummings

1926

since feeling is first
who pays any attention
to the syntax of things
will never wholly kiss you;
wholly to be a fool
while Spring is in the world

my blood approves,
and kisses are a better fate
than wisdom
lady i swear by all flowers. Don't cry
—the best gesture of my brain is less than
your eyelids' flutter which says

we are for each other: then
laugh, leaning back in my arms
for life's not a paragraph

And death i think is no parenthesis
1. The Perl Philosophy
----------------------

`perldoc` -------

example of state keyword in "perldoc -f state" search

```
state EXPR
state TYPE EXPR
state EXPR : ATTRS
state TYPE EXPR : ATTRS

"state" declares a lexically scoped variable, just like "my". However, those variables will never be reinitialized, contrary to lexical variables that are reinitialized each time their enclosing block is entered. See "Persistent Private Variables" in perlsyn for details.
```

perl v5.16.1                      2012-10-01                     0N6GYFHQKB(1)

```
code:
```

[hood@panix2 ~/perl 10:02:19]$ cat q7
use v5.16.1 ;

sub f {
  state $f = 10;
  say $f++ ;
}

f for 1 .. 5;

[hood@panix2 ~/perl 10:02:33]$ perl q7
10
11
12
13
14

"
2. Perl and Its Community

CPAN, perlbrew, github; gitpan, irc
expressivity
------------

DWIM, "Do What I Mean" principle
Perl as a language with many ways of expressing the same thing,
with a number of "linguistic shortcuts".

```
use v5.16.1 ;

my @numbers = (2, 4, 6, 8);
my @trip = map {$_ * 3 } @numbers ;
say for @trip ;
```
context
-------

governs the amount as well as the kind of data to use.

Amount context: void, scalar and list context

"
Value context: Numeric, String and Boolean Context

```
use v5.16.1;
my $p = 'a';
say $p++ for 1 .. 5;
say $p+20;

[hood@panix2 ~/perl 10:41:33]$ perl num.str.context.1
a
b
c
d
e
20
```

implicit ideas
-------------

default scalar variable, aka "topic variable"
$_ (pronoun "it")
& default array variable @_ (pronoun "they")

a). default scalar

```
[hood@panix2 ~/perl 11:03:14]$ cat def.scalar.var.1
use v5.16.1 ;
$_ = "back it up, flip it and reverse it";
say ;
$_ = reverse ;
say ;
say reverse ; # ???
say scalar reverse ;
```

```
[hood@panix2 ~/perl 11:03:53]$ perl def.scalar.var.1
back it up, flip it and reverse it
ti esrever dna ti pilf ,pu ti kcab
back it up, flip it and reverse it
```

"say imposes list context on its operands.
reverse passes its context on to its operands,
treating them as a list in list context and a
concatenated string in scalar context."
b). default array

"[hood@panix2 ~/perl 11:22:09]$ cat def.array.1
use v5.16.1 ;
sub f {
    say "<@_>
}
    f( 1 .. 5 );
f( reverse 1 .. 5 );

sub g {
    wantarray ? "all <@_>" : "only one: " . shift ;
}
    say g(15) ;
say scalar g(15) ;

[hood@panix2 ~/perl 11:22:24]$ perl def.array.1
<1 2 3 4 5>
<5 4 3 2 1>
all <15>
only one: 15
"
2. Digression, into parsing, lex & bison, grammars

```
[hood@panix2 ~/perl/l2 12:06:37]$ cat calc.l
%
#define YYSTYPE double
#include <stdio.h>
#include "y.tab.h"

double yylval ;
%
%
[-][0-9][.][0-9]  yylval=atoi(yytext);
return VL ;
"("    return OP ;
"="    return PL ;
"-"    return MI ;
="/"    return DV ;
"*"    return MU ;
")"    return CL ;

//;
"n    //;
```

%{
#define YYSTYPE double
#include <stdio.h>
#include <string.h>

void yyerror(const char *s) {
    fprintf(stderr, "error %s\n", s);
}
int yywrap() { return 1; }
%
%
token VL OP PL MI DV MU CL
%%

commands: //
    | commands expr { printf("%f\n", $2); }
    ;

mul:      mul VL        { $$ = $1 * $2 ; }
    | base
    ;

sum:      sum VL        { $$ = $1 + $2 ; }
    | base
    ;

base:     expr
    | VL
    ;

expr:     OP PL sum sum CL      { $$ = $3 + $4 ; }
    | OP MU mul mul CL      { $$ = $3 * $4 ; }
    | OP MI base base CL    { $$ = $3 - $4 ; }
    | OP DV base base CL    { $$ = $3 / $4 ; }
    ;

%

int main()
{
    yyparse();
}
}
[hood@panix2 ~/perl/l2 12:04:48]$ lex calc.l
[hood@panix2 ~/perl/l2 12:05:51]$ yacc -d calc.y
yacc: 2 reduce/reduce conflicts.
[hood@panix2 ~/perl/l2 12:06:00]$ gcc -o clc lex.yy.c y.tab.c
[hood@panix2 ~/perl/l2 12:04:00]$ ./clc
(* (- 3 (+ 5 3)) (- 3 (+ 5 3)) )
25.000000
(* (- 3 (+ 5 3)) (- 3 (+ 5 3)) 2)
50.000000
(* (- 3 (+ 5 3)) (- 3 (+ 5 3)) 3 5 )
375.000000
(* (- 3 (+ 5 3)) (- 3 (+ 5 3)) 3 5 13 53 98 )
25320750.000000
(* (- 3 (+ 5 3)) (- 3 (+ 5 3)) 3 53 (+ 20 20))
error syntax error
[hood@panix2 ~/perl/l2 12:06:55]$
Type 1, Context-Sensitive
no rules on left side consists of more symbols than on the right

Type 2, Context-Free
can only have one non terminal on left side

Type 3, Regular
can only have one non terminal on right side
A parser is responsible for identifying tokens and making some form of logical sense from them. Before examining how exactly to accomplish this, let’s first acknowledge that token identification alone is far from adequate. Consider the following statement:

Perl about book parsing is This

While you can easily identify the different tokens found within the statement, it is nonsensical in its current form, thereby illustrating why the rules that govern parsing are considered to be grammars. The statement makes much more sense written in the following form:

This book is about Perl parsing

When performing parsing tasks, token order is quite important and can often affect the meaning or significance of a given token. For instance, a math equation consists of operands and operators, while a sentence consists of nouns, verbs, adjectives, and so forth. In both cases, the order in which their components are written creates the context. To understand this concept, consider the following two sentences:

I saw a bird fly by my window.

A fly landed on my wall.

Within the first sentence, the token fly operates as a verb since it has an action associated with it. However, in the second sentence, the same token operates as a noun. For this reason, most advanced parsing tasks utilize grammars.

In computer science, grammars serve the same purpose as they do for any spoken or written language; they define the orders that tokens can be arranged and thereby help to specify the meaning of the tokens. If the input meets the criteria laid forth by the grammar, then the parser should be able to assign a meaning to the various input tokens. If the grammar criteria are not met, the parser will not be able to make sense of the input, as in the “Perl about book parsing is This” statement. When thinking about parsers in this context, consider how your Perl compiler operates, since much of a compiler’s underlying
function is the ability to parse computer code for a given programming language. If your code follows proper Perl syntax (grammar), then the program will compile and execute. If syntax errors are present, the Perl compiler loses its ability to properly interpret your code, and you will receive a runtime error message similar to the following:

Syntax error at MyPerlScript.pl at line 5, near "xxxx"
Execution of MyPerlScript.pl aborted due to compilation errors.

Tip Many books about compiler design can provide highly beneficial resources for furthering your knowledge of parser design and theory. One recommended resource is *Compilers* by Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman (Addison-Wesley, 1986).

Now that you have an idea of what a grammar is supposed to accomplish, let’s look at how you can implement such functionality; I will now discuss the computer scientist’s view of grammars.

**Introducing Generative Grammars**

In the past, many linguists tried to define grammars that were exact, were finite-sized, and provided a complete description of the language (that is, the allowable sequences of tokens). However, this type of construction led to many problems; in fact, arriving at a complete description that could be proven to hold true against all possible cases in a given language was determined to be a nearly impossible task because of the potentially infinite numbers of combinations of tokens. A good example of this is the text contained within this book, or any book. Every sentence in this book, with the exception of the code listings and the purposely incorrect examples shown previously, adheres to what is considered proper English grammar, yet no two sentences consist of the same combination of words given in the same order. Indeed, it would be difficult to express a finite-sized set of rules that would perfectly match every sentence, especially considering that English, and most other languages, are replete with context-sensitive exceptions. For example, let’s take the simple rule of making a verb past tense by adding *ed* at the end so that *walk* becomes *walked*. So far, so good, but what about the verb *run*? In the past tense it is not *runned*; it is *ran*. Therefore, a special rule would need to be established just for this word. Now consider just how many other exceptions there are, how many words there are, and how many different possible ways these words can be combined, and take a moment to ponder how immense a task developing a complete description of a language would be.

Thus, the complete description idea has largely been abandoned, and generative grammars have come into play. Generative grammars do not seek to provide complete descriptions of a language, but rather they specify an exact, fixed-sized recipe for
constructing sentences (sequences of tokens) in a given language. In other words, a generative grammar may not be able to be used to create every possible sentence in a given language, but it does guarantee that any sentence generated with it will be in a proper form. This view of grammars has been determined to be more widely applicable and is quite the same view of grammars that computer scientists embrace.

Grammar Recipes

The recipe-based grammar method is based upon the simple principle that a properly formed sequence of objects can be built by starting with a small object and using rules to add objects to it and create a larger structure. This is an underlying principle for all the grammar types discussed in this chapter, since even though the various types of grammars have differing levels of restrictions, they are all generative in nature. To clarify what exactly this entails, let’s look at a simple recipe that describes a list of items you might want to pick up at the grocery store (for example, bread, milk, and eggs). Thinking back to grammar-school English classes, you probably recall learning the basic rules for constructing such a list, such as each item other than the second-to-last item and the last item should be preceded by a comma, and there should be an and present between the last item on the list and the preceding item, if any. According to these rules, the list should look something like this:

\[
\text{Item, Item, Item, ... Item and Item}
\]

**Note** In the English language, it is also acceptable to have a comma present before the and that separates the last item from the second-to-last item, but for purposes of this chapter I will treat the syntax shown in the previous example as the only acceptable syntax.

In this example, I will slightly increase the specificity of the list and generate a recipe that can be used to describe grocery lists containing any of the following items: bread, milk, eggs, and meat. Thus, how do you use the information you know about the list structure to create such a recipe? A logical starting point is to decide that bread, milk, eggs, and meat are all items and thus can replace any of the Item placeholders present in the list. Next you should consider the simplest possible list structure. That is a list that contains only one Item. Therefore, in this simple grammar, any one item can serve as a complete sentence (sequence of tokens). Longer lists can be created by appending Item to the end of a simpler list. It is also possible for each of these longer lists to serve as a complete sentence, in which the end of the list signifies the end of the sentence. However, to achieve the proper format for the ending element of the list, you must also specify a rule that when Item is found at the end of a sentence, it must be replaced by and Item.
Now that you have a general understanding of the grammar for a mini “list” language, how do you represent these thoughts in the form of a more formalized grammar? Writing out such a grammar should result in something like this:

Item → bread|milk|eggs|meat
Sentence → Item|List END
List → Item|Item, List, Item END → and Item

In this particular notation, tokens that begin with a capital letter are nonterminals, and tokens that begin with a lowercase letter are terminals. Terminals are items that cannot be further specified. That is, they cannot be made any more specific. In the previous example, bread is a terminal because this grammar considers bread to be a fully specified item; it does not further differentiate between white bread and whole wheat per se. If bread were instead a nonterminal, the grammar would need the following modifications:

Item → bread|milk|eggs|meat
Bread → white|whole wheat

Nonterminals differ in that they can be replaced by even more specific components. For example, the nonterminal Item can be replaced by any of the terminals eggs, bread, milk, or meat. It is also possible to replace nonterminals with other nonterminals, as in the List nonterminal being replaced by the Item nonterminal or the Item, List nonterminals. This ability to replace List with Item, List is especially important since it allows the grammar to be used for generating a list of any length rather than requiring an individual rule for each list of length 2 to length N.

Also in this particular notation, the -> symbol stands for “can be replaced by,” and the subscript S defines the start symbol for the sequence. The start symbol is important since it represents the point at which sentence generation will begin. Chapter 3 will also show that top-down parsing tasks are initiated at the start symbol, and bottom-up parsers try to work their way back from a complete sentence to the start symbol. Furthermore, all grammars must contain a finite number of terminals, a finite number of nonterminals, and a finite number of production (grammar) rules. Additionally, a start symbol must always be specified in order for a grammar to be fully defined.

---

**Note** Do not get overly caught up in the grammar syntax used in the previous example; it is being used largely for illustrative purposes. Most Perl parsers instead require that grammars be specified in the Backus-Naur format (BNF), which is syntax used to specify a higher-order grammar than that found in the preceding example. I will discuss this format in the upcoming “Backus-Naur Format” section.
Sentence Construction

Now that you have specified a grammar that defines the syntax of an item list language, it is time to put the grammar to the test and determine if it can really be utilized to construct an appropriate list sentence. Therefore, let’s begin to sequentially step through the rules laid forth and try to construct a list using only the rules provided by the grammar.

To begin constructing the grocery list sentence, you first call upon the rule with the start symbol, which means you have a new sentential form that consists of the nonterminal Sentence. You can think of a sentential form as a “sentence in progress,” since it is a form that contains nonterminals. To construct a fully qualified sentence, you must replace all nonterminals with terminals. Table 2-1 lists the order in which the various rules need to be called and the results they will produce.

<table>
<thead>
<tr>
<th>Sentential Form</th>
<th>Rule Used*</th>
</tr>
</thead>
<tbody>
<tr>
<td>List END</td>
<td>Sentence -&gt; List END</td>
</tr>
<tr>
<td>Item, List END</td>
<td>List -&gt; Item, List</td>
</tr>
<tr>
<td>Item, Item, List END</td>
<td>List -&gt; Item, List</td>
</tr>
<tr>
<td>Item, Item, Item, List END</td>
<td>List -&gt; Item, List</td>
</tr>
<tr>
<td>Item, Item, Item END</td>
<td>List -&gt; Item</td>
</tr>
<tr>
<td>Item, Item and Item END</td>
<td>, Item END -&gt; and Item</td>
</tr>
</tbody>
</table>

* The Rule Used column indicates the rule used to create the current sentential form from the previous one.

After establishing that the Sentence nonterminal is going to serve as a start symbol, you can next check to see what rule establishes the valid replacements for Sentence. Looking at the rules established earlier, you see that the nonterminal Sentence can be replaced by the more specific nonterminal List. Thus, by applying this rule, your sentence now consists of List END. You then repeat the process of increasing the specificity of the sentential form by determining what can replace the nonterminal List. In this case, since you want a list of four items, you then apply the rule that List can be replaced by Item, List. You then apply this rule twice more, which yields the following:

Item, Item, Item, List END

Since there is only one list element left to add to the list of four, you then call upon the rule that states List can be replaced by Item in order to eliminate the final List non-terminal, yielding the following:

Item, Item, Item, Item END
You next seek to eliminate the END marker and insert and between the final two items on the list. Thus, you call upon the rule, Item END -> and Item to accomplish this. At this point, you are left with the following:

Item, Item, Item and Item

As a final step, you call upon the rule, which specifies acceptable values for the non-terminal items, and replace those nonterminals with the terminals eggs, milk, bread, and meat. This yields the following sentence:

eggs, milk, bread and meat

To make the preceding process easier to follow, Figure 2-1 demonstrates a production graph (also referred to as a syntactic graph by some). The diagram illustrates the syntactic structure used to generate the final sentence.

---

**Note** The grammar just discussed is merely one example of how that grammar could be used to generate a valid list sentence. Sentences such as “bread” and sentences such as “bread and eggs” and “eggs, eggs and eggs” would also be valid sentences that could be reproduced according to the grammar provided.

---

**Introducing the Chomsky Method**

The type of grammar used to generate the list sentence is referred to as a phase-structure grammar. In theory, you can use this type of grammar to lay out the rules for any type of set (sentence) structure. However, as the sentence grows in complexity, so too does the number and types of rules in the grammar. Thus, for complex structures, the design and implementation of a phase-structure grammar can become quite arduous. When performing parsing tasks, the difficulty of working with phase-structure grammars is compounded even further, since no generalized parsing algorithm exists for use with them. More restrictive grammars (most notably Type 2 grammars) lend themselves to being parsed into treelike structures, which can be readily processed using various programming methodologies. As you will see in the upcoming “Type 2 Grammars (Context-Free Grammars)” section, creating a parse tree requires that the rules of the grammar are not subject to context sensitivity/dependency. Context sensitivity implies that the meaning of a nonterminal or its proper replacement during sentence generation can be deciphered only by looking at the other nonterminals and terminals surrounding it. A phase-structure grammar’s unrestricted nature also allows for the simultaneous replacements of multiple nonterminals and terminals, which also leads to difficulties in their implementations. This lack of restrictions and context dependence creates a lot of “specialized cases” that parser code needs to deal with on an individual basis. Therefore, specialty parsers need to be coded for use with
them, but even this is not always possible and is usually a complex and inefficient process. For this reason, these so-called Type 0 grammars are generally not employed in the real world; instead, a more restricted type of grammar is used for most applications.

**Note** The notable exception to where phase-structure grammars or the upcoming context-sensitive grammars are sometimes used is in the field of linguistics. Representing natural language with a more restrictive grammar is sometimes not feasible given the large numbers of exceptions and caveats present in most modern-day spoken and written languages. The difficulty of implementing phase-structure grammars programatically is a major reason why computer processing of natural language is such a difficult and elusive task.
These shortcomings are addressed in the Chomsky hierarchy of grammars, which seeks to increase the manageability of grammars while still preserving the robustness of the grammar’s descriptive abilities. The hierarchy was first proposed by Noam Chomsky in 1942. Under the Chomsky hierarchy, phase-structure grammars are considered Type 0 grammars, since they exist in a fully unrestricted form. The hierarchy has three other types of grammars (Types 1–3), each with increased restrictions over the previous but also with increased manageability. Fortunately, even the more restrictive cases are still quite applicable to many parsing tasks. In fact, some of the most popular Perl parsing modules, such as RecDescent (Chapter 5), utilize Type 2 grammars.

**Type 1 Grammars (Context-Sensitive Grammars)**

If you look back at the Type 0 grammar you constructed for the list language, you will note that the grammar allowed you to create the following rule:

\[
\text{Item END} \rightarrow \text{and Item}
\]

In this rule, you are able to replace three symbols (comma, Item, and END) with just two symbols (and and Item). In a Type 1 grammar, such replacements are unallowable. A grammar is considered context-sensitive if only one symbol gets replaced by one or more other symbols.

---

**Note** Type 1 grammars are often referred to as *monotonic grammars*, since a monotonic grammar is defined as a grammar that contains no rules in which the left side consists of more symbols than the right side. This means, by definition, all context-sensitive grammars are also monotonic.

---

Hence, if you had to rewrite the rules of the list language to adhere to the standards of a Type 1 grammar, you would write the following:

\[
\begin{align*}
\text{Item} & \rightarrow \text{eggs|milk|bread|meat} \\
\text{Sentence} & \rightarrow \text{Item|List} \\
\text{List} & \rightarrow \text{EndItem|Item Comma List} \\
\text{Comma EndItem} & \rightarrow \text{and EndItem} \\
\text{and EndItem} & \rightarrow \text{and Item} \\
\text{Comma} & \rightarrow ,
\end{align*}
\]

Note how you have to add two nonterminals (Comma and EndItem) and two rules in order to conform to the context-sensitive standard. This prevents the potential complications that could be associated with the simultaneous multiple replacements (see Figure 2-2).
I will explain the construction of Type 1 grammars a little more closely now using the standard computer science example of generating a string that consists of an equal number of a’s, b’s, and c’s. In other words, say you want to define a grammar that will accept the string abc and expand upon it to generate strings such as aabbcc and aaaaabbbcccc.

**Figure 2-2.** The Type 1 production graph for the “eggs, milk, bread and meat” sentence. The regions where the arrows cross are regions of context sensitivity.
To begin this grammar, you start with the most straightforward rule, in that Sentence could be replaced by abc. The question arises, however, of how you append things to this string. Since the grammar you are trying to create is a Type 1 grammar, you cannot readily create rules for complex insertions of new characters; therefore, you will create a rule that adds a to the beginning of the string and appends a nonterminal X to the end of the string. This yields a rule of the following form:

Sentence \rightarrow abc | aSentenceX

Since this rule directly takes care of adding a to the beginning of the string, you need to add b and c by creating a rule that states the nonterminal X can be replaced by bc. However, directly appending bc to the end of the string would not give you the result you desire. Thus, you must first create a rule that strategically positions X prior to its replacement with its corresponding terminals. You can do this as follows:

\[
\begin{align*}
cX & \rightarrow Xc \\
\end{align*}
\]

Applying this rule one or more times will eventually result in a sentential of the form bXc. Now if you perform your replacement of X with bc, both the b and c will be appropriately placed. Therefore, the resultant grammar for the string expansion is as follows:

\[
\begin{align*}
\text{Sentence} \rightarrow & \ abc | a\text{Sentence}X \\
cX \rightarrow & \ Xc \\
bXc \rightarrow & \ bbcc \\
\end{align*}
\]

If you examine the generative nature of this grammar in constructing aabbcc, you will see that the procedure is as follows. First, you apply the following rule:

\[
\text{Sentence} \rightarrow \ a\text{Sentence}X
\]

Following this, Sentence can be replaced according to this rule:

\[
\text{Sentence} \rightarrow \ abc
\]

which yields the following:

\[
aabbXc
\]

Now if you apply this rule:

\[
cX \rightarrow Xc
\]

you arrive at this:

\[
aabXc
\]
Executing the remaining rule, which dictates the replacement of $X$ by $bc$, you are left with the desired string (see Figure 2-3).

If you consider each of the previous examples, you will notice that even though only one nonterminal gets replaced in each rule, multiple terminals and nonterminals are still allowed on the left side of each rule. This is what makes these grammars context-sensitive, because a given nonterminal cannot always be replaced with an element on the right side of the rule. Instead, the complete left side of the rule must match so that the substitution occurs only in the event that the context in which the nonterminal is found is proper. Context cannot always be readily determined, however, and with the creation of a larger set of rules, most grammars can be made into context-free grammars (see the next section), which are the preferred type of grammars for parsing tasks. As you will see in the next chapter, the most common classes of parsing algorithms are designed to work with context-free grammars as well.
Type 2 Grammars (Context-Free Grammars)

Grammars of this type are probably most important for dealing with Perl-based parsing routines, since most of the common Perl parsing modules will utilize Type 2 grammars. What distinguishes Type 2 grammars from Type 1 grammars is that Type 2 grammars are considered *context-free* instead of *context-sensitive*. In more technical terms, this means that while context-sensitive grammars are allowed to have one or more nonterminals on their left side, context-free grammars have the additional restriction of being able to have only a single nonterminal on their left side. For example, if you were to rewrite your list language in the form of a Type 2 grammar, it would take the following form:

Item → bread|milk|eggs|meat
Sentence_s → Item|List and Item
List → Item, List|Item

The significance of obtaining a grammar in a Type 2 format is that since it is context-free, each nonterminal is able to operate independently of its neighbor. In other words, each grammar rule operates as a definition of the element found on the left side. This yields the benefit that the production graph produced will always be in a tree form, and hence Type 2 production graphs are often referred to as *production trees* (see Figure 2-4). The benefit to a tree-type grammar is that the sentences constructed from its rules can be analyzed by both top-down and bottom-up approaches. As you will see in the next chapter, this is ideal for parsing tasks, since recursive descent parsers employ the top-down approach, and shift-reduce parsers utilize the bottom-up approach. Type 2 grammars are probably the most widely used of all the grammar types when it comes to creating parsers, since they allow the data to be parsed into treelike structures. Tree-based analytical methods are quite pervasive throughout computer science. As you will see in upcoming chapters, tree-based parsing is utilized in processing a diversity of data formats, including HTML and the DOM-based parsing of XML documents.

Backus-Naur Format

BNF is named after John Backus and Peter Naur, who were two prominent computer scientists in the area of compiler design in the mid-twentieth century. BNF was originally created as a way of specifying the syntax of the ALGOL programming language but has become one of the standard notations for expressing grammars in computer science. In Perl, grammars are generally structured using BNF. The previous rules written out in BNF would look as follows:

```plaintext
<Item>::=       bread|milk|eggs|meat
<Sentence> ::=  <Item>|<List> and <Item>
<List> ::=       <Item>, <List>|<Item>
```
Before examining the grammar, let's first examine some of the notations used in BNF:

- Symbols that are encapsulated within a set of <>; BNF notation are nonterminals; the tokens that are not enclosed in <> are terminals.

- The ::= in BNF notation stands for “may be replaced by,” and | is used for alternation of choices in a manner similar to the Perl regular expression syntax.

- The subscript S denotes the start symbol for the sequence, and the start symbol must always be a nonterminal.
AN ALTERNATE SYNTAX FOR TYPE 2 GRAMMARS

Although not widely used by Perl programmers, programmers in other languages sometimes express Type 2 grammars in van Wijngaarden form. This format is as follows:

Item: bread symbol; milk symbol; eggs symbol; meat symbol.
Sentence: Item; List comma symbol, Item.
List: comma symbol, List; Item.

Note the use of ; for alternation and the use of symbol after all nonterminals. A comma (,) also has the special meaning of “followed by,” and all rules must end in a period.

Constructing Math Equations with Type 2 Grammars

I will begin the explanation of Type 2 grammars by starting with a practical example, that of creating grammar rules that can be used to construct mathematical equations. In particular, I will show you how to create algebraic expressions employing this format:

Variable = Expression

For example:

\[ w = x*(y+z) \]

Now that you have an idea in mind of what you want to accomplish, let’s look at how you could go about constructing such an equation via a Type 2 grammar.

\[
\begin{align*}
\text{Variable} &::= w|x|y|z \\
\text{Sentence} &::= \text{Variable}=\text{Expression} \\
\text{Expression} &::= \text{Expression}+\text{Expression} \\
\text{Expression} &::= \text{Expression}-\text{Expression} \\
\text{Expression} &::= \text{Expression}*\text{Expression} \\
\text{Expression} &::= \text{Expression}/\text{Expression} \\
\text{Expression} &::= (\text{Expression}) \\
\text{Expression} &::= \text{Variable}
\end{align*}
\]

The previous grammar defines the basic rules essential to constructing fundamental four-function algebraic expressions and will now be put to the test by using it to generate the equation for \( w \) presented previously.

Table 2-2 summarizes the order in which the rules are applied to generate the desired algebraic expression, and Figure 2-5 demonstrates the production tree for the process.
# Table 2-2. Application Order of Algebraic Grammar Rules

<table>
<thead>
<tr>
<th>Sentential Form</th>
<th>Rule Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>V=E</td>
<td>S::= V=E</td>
</tr>
<tr>
<td>V=E*E</td>
<td>E::= E*E</td>
</tr>
<tr>
<td>V=E*(E)</td>
<td>E::= (E)</td>
</tr>
<tr>
<td>V=E*(E+E)</td>
<td>E::= E+E</td>
</tr>
<tr>
<td>V=V*(V+V)</td>
<td>E::= V (applied three times)</td>
</tr>
<tr>
<td>w=x*(y+z)</td>
<td>V::= w</td>
</tr>
</tbody>
</table>

*To improve readability, $S$, $V$, and $E$ were used to replace `<Sentence>`, `<Variable>`, and `<Expression>`, respectively.*

---

![Production Tree](image)

**Figure 2-5.** The production tree for the construction of the equation $w=x^*(y+z)$
Constructing Basic English Sentences

You may recall from middle-school grammar class that a complete sentence in the English language usually has a subject, an object, and a verb, and they are usually present in the order subject, verb, object. To create a slightly more advanced sentence, you may choose to insert an adjective before each noun. So, now the question arises as to how you express this basic sentence structure in the form of a Type 2 grammar. To examine this issue, let's review the following Type 2 grammar:

\[
\begin{align*}
\text{Sentence} &::= \text{Subject} \ \text{Verb} \ \text{Object} \\
\text{Subject} &::= \text{NounExpression} \\
\text{Object} &::= \text{NounExpression} \\
\text{NounExpression} &::= \text{Noun} | \text{Adjective} \ \text{NounExpression} \\
\text{Noun} &::= \text{grammars} | \text{parsers} \\
\text{Adjective} &::= \text{Type 2} | \text{Backus-Naur} | \text{better} | \text{Perl} \\
\text{Verb} &::= \text{make}
\end{align*}
\]

Now, if you step through these given rules in the order presented in the production tree shown in Figure 2-6, you arrive with this English sentence:

Type 2 Backus-Naur grammars make better Perl parsers

However, when constructing English sentences via a grammar, it is important to remember that the limitations imposed by their context-free nature can arise. For example, if you had the following grammar:

\[
\begin{align*}
\text{Sentence} &::= \text{Subject} \ \text{Verb} \ \text{Object} \\
\text{Subject} &::= \text{NounExpression} \\
\text{Object} &::= \text{NounExpression} \\
\text{NounExpression} &::= \text{Noun} | \text{Adjective} \ \text{NounExpression} \\
\text{Noun} &::= \text{cats} | \text{bird} \\
\text{Adjective} &::= \text{two} | \text{one} \\
\text{Verb} &::= \text{chased}
\end{align*}
\]

you could use it to create the following proper English sentence:

two cats chased one bird

However, it could also easily generate the incorrect sentence:

one cats chased two bird

This is an example of how the context-free nature of the Type 2 grammar makes the parser unable to correctly generate sentences where one component affects the
appropriateness of another (for example, one should go only before the singular bird). In this case, you can do several things to remedy the matter, with the first being using a context-sensitive or phase-structure grammar instead. This, however, tends to become an arduous process, since both grammars seek to define global relationships through localized dependencies. Programmers more often choose to create a “context fixing” program instead to fix the output after its context-free grammar generation. In fact, this type of complication is one of the reasons why certain instances of machine translation pose a high degree of difficulty. Luckily, most parsing tasks deal with data sets that are much more finite and regularly structured than human language, and hence routine parsing tasks generally do not entail these complications.

**Tip** Generally speaking, it is best to use the simplest grammar structure you can when solving a problem, as long as it does not negatively influence the robustness of the parser’s ability to process data. By using a simpler grammar, you will generally get improved execution speed as well as less of a headache in actually developing the parser.
Extended Context-Free Grammars

Extended context-free grammars are a system of notation designed to improve the readability of grammars that require repetition. For example, in the English sentence examples given previously, the grammar is capable of looping through several times to yield multiple adjectives before a noun is placed.

\[
\text{<NounExpression>::= <Noun}|\text{<Adjective> <NounExpression>}
\]

Rather than writing it in this format, an extended context-free grammar would instead allow you to write this:

\[
\text{<NounExpression>::= <Noun}|\text{<Adjective>+ <Noun>}
\]

or an equivalent type statement. The plus (+) sign in this case is a repetition operator and indicates the allowable types of repetition. The + symbol means one or more repetitions, an asterisk (*) means one or more repetitions, and a question mark (?) indicates zero or one repetitions.

Type 3 Grammars (Regular Grammars)

Up until this point, I have dealt with restrictions placed on the left side of a given rule. What distinguishes Type 3 grammars from Type 2 grammars is a right-side restriction. Type 3 grammars are allowed to have only one nonterminal on the right side of any given rule. This restriction means that in order for any form of sentence generation to occur, nonterminals must always be present at the end of a sentential form. For example, if you were to create a Type 3 grammar for your list language, you would write something like the following:

\[
\text{Sentence-> [bread|milk|eggs|meat]|List}
\]

\[
\text{List-> [bread|milk|eggs|meat], List|[bread|milk|eggs|meat] and [bread|milk|eggs|meat]}
\]

This creates a production chain type of construction, as shown in Figure 2-7.

Despite the restrictions placed on Type 3 grammars, they are actually not without real-world value and can be used for many simple parsing tasks. In fact, anyone who worked their way through Chapter 1 before reading this chapter has actually already employed Type 3 grammars, since such grammars form the basis of regular expressions. When you create a regular expression, you are in essence creating a Type 3 grammar that Perl’s regex engine uses to parse strings of text.

---

**Note** Keep in mind, though, that authoring generators and parsers for Type 3 grammars is the simplest, so if a Type 3 grammar is sufficient, it is advisable to use it.
Another type of restriction is not part of the formal Chomsky hierarchy but still has occasional uses. A finite-choice grammar has the additional restriction that only terminals are allowed on the right side of the rule. You can use this type of grammar to represent a listing of keywords. Finite choice rules are often seen as pieces of larger, less restricted grammars (for example, \texttt{<Item>::= milk|eggs|bread|meat}).

**Tip** Another type of restriction is not part of the formal Chomsky hierarchy but still has occasional uses. A finite-choice grammar has the additional restriction that only terminals are allowed on the right side of the rule. You can use this type of grammar to represent a listing of keywords. Finite choice rules are often seen as pieces of larger, less restricted grammars (for example, \texttt{<Item>::= milk|eggs|bread|meat}).

---

### Using Perl to Generate Sentences

Although the focus of this book is on parsing existing sentences and data structures, I will digress from this point momentarily to explain how you can utilize grammars, like the ones constructed previously, to generate sentences. Sentence generation is often important when it comes to transforming one document’s contents into another format. For example, XML documents are often translated via the rules defined in an XML schema from one format into another so applications can communicate without having to be able to explicitly read the file formats of another application. This not only should give you a better appreciation of the expressive power of grammars but also provides a way in which you can test your grammar for certain common caveats.
Perl-Based Sentence Generation

Now it is time to actually implement one of the context-free (Type 2) grammars in a Perl script and see how well it functions. As a starting point, let’s consider your list language grammar one more time and begin by entering the following grammar into a text file:

\[
\text{<Sentence>} ::= \text{<Item>} | \text{<List>} \quad \text{and} \quad \text{<Item>}
\]
\[
\text{<List>} ::= \text{<Item>}, \text{<List>} | \text{<Item>}
\]
\[
\text{<Item>} ::= \text{bread} | \text{milk} | \text{eggs} | \text{meat}
\]

It is important to maintain the spacing shown in order to get the following script to execute as is, since you will use regular expressions to parse through the grammar and since for simplicity purposes code was not added in order to eliminate extraneous whitespace. Listing 2-1 shows the script you can utilize to test this grammar.

**Listing 2-1. A List Generator**

```perl
#!/usr/bin/perl

my @rules;
my $SententialForm;
my @rhs;

# read in grammar rules file
while(<>){
    push @rules, [split/::=/];
}

#Sets sentential form equal to <Sentence>
$SententialForm=$rules[0][0];

#While checks for remaining nonterminals
while($SententialForm=~/<\w*>/) {
    $NonTerm=$&;

    #Loops through rules to find nonterminal matches
    foreach $rule (@rules){
        $lhs = $rule->[0];
        $rhs = $rule->[1];
        if($NonTerm=~/$lhs/){
            #If rule matches sets, nonterminal equal to rule rhs
            if($rhs!~/\|/){
                $SententialForm=~s/$NonTerm/$rhs/;
            }
        }
    }
}
```

CHAPTER 2 □ GRAMMARS
If rule has more than one rhs, picks one at random
elsif($rhs=~/\|/){
    # @rhs is an array made of the alternate elements of $rhs
    @rhs=split \|/, $rhs;

    # replace $NonTerm with a random element of @rhs
    $SententialForm=~s/$NonTerm/$rhs[int(rand(@rhs))]/;
}
}

When no nonterminals are remaining, prints generated sentence
print $SententialForm;

The routine begins by using a while loop to read in the rules of your grammar from a
file, which is passed to the script as an argument. The split function is utilized to divide
the rules according to their left and right divisions, and the strings corresponding to each
of these sides are stored in the array rules.

# read in grammar rules file
while(<>){
    push @rules, [split/:= /];
}

Next this script assumes that the first rule of the grammar is the start rule and
accordingly sets the first sentential form to the nonterminal on the left side of this rule,
which in this case is <Sentence>.

#Sets sentential form equal to <Sentence>
$SententialForm=$rules[0][0];

You next arrive at a loop structure that checks the sentential form for nonterminals,
since a complete sentence has been generated once a sentential form containing only
terminals has been achieved. If a nonterminal is detected, the variable $NonTerm is set as
the leftmost nonterminal value.

#While checks for remaining nonterminals
while($SententialForm=~/<\w*>/){
    $NonTerm=$&;
}

When dealing with context-free grammars, you can perform successive nonterminal
replacements in a left-to-right or a right-to-left manner and still end up with equivalent
results, but since Perl regular expressions operate in a left-to-right manner, this approach is much easier to code.

Now that you have identified a nonterminal, it is time to cycle through the rules of your grammar in order to locate the rule that specifies what the nonterminal can be changed to. You can accomplish this by comparing the nonterminal to the left-side values stored in the @rules array. Once a match is found, it becomes time to work with the right side of the rule. Conditional statements are used to evaluate whether alternation is present within the right side of the matching rule. If alternation is not present, a substitution is performed to replace the nonterminal with the right side of the rule.

```perl
#Loops through rules to find nonterminal matches
foreach $rule (@rules){
    $lhs = $rule->[0];
    $rhs = $rule->[1];
    if($NonTerm=~/$lhs/){
        # If rule matches sets, nonterminal equal to rule rhs
        if($rhs!~/\|/){
            $SententialForm=~s/$NonTerm/$rhs/;
        }
    }
}
```

**Note** This routine assumes that the same nonterminal does not exist on the left side of a rule more than once.

If alternation is detected, the right side of the rule is split into its component options and the values stored in the array @rhs. The script then picks a random choice from the @rhs array by successively applying the rand and int functions to the array. In this way, a random integer is used to pick the piece of the right side that will replace the nonterminal in question. To more thoroughly test a grammar, a breadth-first search or depth-first search is often used to produce every possible sentence, but such exhaustive generation is not necessary to demonstrate how the concept works.

```perl
elsif($rhs=~/\|/){
    # @rhs is an array made of the alternate elements of $rhs
    @rhs=split /\|/, $rhs;

    # replace $NonTerm with a random element of @rhs
    $SententialForm=~s/$NonTerm/$rhs[int(rand(@rhs))]/;
}
```
This process will be cycled through until all nonterminals have been replaced by sentences, and at this point the completed sentence will be output. For example, I achieved the following results when I ran this script three times:

meat, milk and eggs
milk
meat, eggs, bread, milk, milk, eggs and eggs

As you can see, they are all valid lists according to the grammar rules specified.

---

**Tip** For those attempting to run this script on Windows, keep in mind that Windows-based text files have a carriage return appended to the end of each line. Unless the carriage returns are removed, they may be substituted into the sentential form along with the right-side text, yielding a correct, but multiline, output.

---

### Avoiding Common Grammar Errors

Among the most common types of grammar errors observed are nonterminals that never break down into anything more meaningful. In other words, you cannot complete your sentence generation, because this type of condition means that one nonterminal can never be converted to a terminal according to the rules defined. This is due to either a missing rule that will add this conversion capability or the presence of a loop condition. Such a loop condition exists in situations like the following:

<NonT1> → <NonT2> → <NonT3> → <NonT1>

Considering the previous condition, you will notice that a rule exists that allows you to move from <NonT1> to <NonT2>, and likewise a second rule exists that allows you to progress from <NonT2> to <NonT3>. The problem, however, occurs when the rule that specifies <NonT3> is encountered, since this rule specifies that <NonT1> should replace <NonT3>. If this is the only path allowed to you in the grammar, your sentence generator will never be able to exit the loop, thereby making sentence generation impossible.

Another common mistake is the absence of a nonterminal from the right side of any rule in the grammar. For example, consider the following grammar:

```
<Sentence>::=<NonT1>|<NonT2>
<NonT1>::=<NonT4>|Terminal1
<NonT2>::=Terminal1|Terminal2
<NonT3>::=<NonT4>|Terminal3
<NonT4>::=Terminal4|Terminal5
```
Within this grammar, $\langle\text{NonT3}\rangle$ is never called from the right side of any rule, which means that $\langle\text{NonT3}\rangle$ will never be called by a parser or generator that employs this grammar.

Although such an absence will not affect execution since its absence means it will never appear in any sentential form, this type of error is still indicative of some form of faulty logic. If a rule exists with the nonterminal on the left side, odds are that the nonterminal was expected to occur in the sentential form at some point during sentence generation. This can often be a frustrating error to locate for large grammars because it is likely hard to visually spot and because it may not always hamper sentence generation.

**Generation vs. Parsing**

As mentioned previously, grammar-based sentence generation can be a useful way for looking for errors within a grammar, but there is also a practical parsing purpose for understanding how this process works. Consider this: When you generate a sentence from a context-free grammar, you basically apply a series of rules in a given order, which in turn generates your production tree and hence your resultant sentence.

Parsing in essence is the reverse of this process (see Figure 2-8). When you perform grammar-based parsing tasks, you start with the resultant sentence; given this sentence and a set of grammar rules, you attempt to reconstruct the production tree. It is through reconstructing the parse tree that you are often able to assign a meaning to the different pieces of your start sentence. This process is called *semantic analysis* and will be covered in the next chapter, where I will discuss the methods employed by both top-down and bottom-up parsers.

![Figure 2-8. The relationship between parsing and generation](image)
Summary

This chapter introduced the idea of a generative grammar and the different types of generative grammars that comprise the Chomsky hierarchy. Of all the Chomsky divisions, Type 2 context-free grammars are the most often utilized for real-world parsing applications, since they can be utilized with most existing parsing algorithms and thus do not require the development of custom parsers. This chapter also provided an overview of how you could utilize a generative grammar in a Perl script to create valid sentences.

In Chapter 3, you will examine the two common classes of parsing algorithms, bottom-up and top-down, and see how you can utilize them in conjunction with a set of grammar rules to parse math equations. Chapter 3 will also touch upon how parsers are utilized in the real world to evaluate content in which the format is not rigidly structured.
Randal Schwartz, "On Parsing Perl", 4, Dec 2000, Perlmonks
On Parsing Perl

by merlyn

on Dec 04, 2000 at 04:21 UTC

{from an alt.perl post I just made, reposted here to solicit feedback from fellow monks...}

>>>>> "Makhno" == Makhno
<Makhno@imakhno.freeserve.co.uk> writes:
Makhno> I'm thinking of writing a GUI Perl-syntax-aware editor, and
Makhno> wondering what's the best way to parse perl? Highlighting
Makhno> reserved words is easy (using, eg, index()) but indentifying
Makhno> things like comments is a bit more difficult.

Makhno> A regex like /#.*/\n/ will catch
Makhno> comments when they are used
Makhno> simply, ie:

Makhno> print "hello\n"; #print hello

Makhno> but will get it wrong when the '#' is
Makhno> used as part of a regex
Makhno> (or in a string)

Makhno> s#hello#goodbye#;
Makhno> print "will behave like a #comment";

Makhno> Does anybody have any ideas on how I go
Makhno> about parsing perl
Makhno> syntax in such a way, before I go to a
Makhno> lot of potentially
Makhno> unnecessary work?

Perl is extremely difficult to parse. In fact, some would say impossible.

One thing that makes it difficult is the dual nature of a half dozen
characters like "/". If that / is being used in a place that's expecting an
operator, it's divide. If it's being used in a place that's expecting an
operand, it's the beginning of a regular expression. So you have to keep
track at all times of whether you're looking for an operator or an operand.

"No problem", you say? Quick... for the following, play the game of "regex or divide?"

```perl
sin / ... 
time / ... 
localtime / ... 
caller / ... 
eof / ... 
```

Got those right? How about these?

```perl
use constant FOO => 35;
FOO / ...
```

```perl
use Fcntl qw(LOCK_SH);
LOCK_SH / ... 
```

OK, and now some of your own:

```perl
sub no_args ()
sub one_arg ($)
sub normal (@)

no_args / ...
one_arg / ...
normal / ...
```

Got those too? How about these (same problem, different file):

```perl
use Random::Module qw(aaa bbb ccc);
aaa / ... 
bbb / ...
ccc / ... 
```

A little harder, eh? So now you have to parse OUTSIDE the file to get your answer. And as if that wasn't enough, let's get weird:

```
BEGIN {
  eval (time % 2 ? 'sub zany ()'; : 'sub zany (@);');
}
```

Quick, was that last one a divide or a regex start?

Why does it matter? Look at this:

```perl
sin / 25 ; # / ; die "this dies!";
time / 25 ; # / ; die "this doesn't die";
```

The first one is computing the sin of the true/false value gotten by matching " 25 ; # " against $_. Then it dies. The second one is computing the time of day divided by 25, then ignoring the comment.

Starting to see the trouble?
This leads people to say "the only thing which can parse Perl (the language) is perl (the binary)". Maybe not for Perl6. But for the Perl we know and can use today, certainly so.

-- Randal L. Schwartz, Perl hacker

Re: On Parsing Perl
by gaggio on Dec 04, 2000 at 04:35 UTC
I don't know if I am with you there. What is the Perl executable doing when it executes a script? Isn't that called parsing also?

Re: Re: On Parsing Perl
by merlyn on Dec 04, 2000 at 04:50 UTC
That's why I said "the only thing that can parse Perl is perl"!

-- Randal L. Schwartz, Perl hacker

Re^2: On Parsing Perl
by Anonymous Monk on May 10, 2008 at 13:03 UTC
Stunning reading comprehension there. I wonder if such clueless folk ever come back and read their comments years later and feel a twinge of embarrassment.

Reaped: Re^2: On Parsing Perl
by NodeReaper on Jul 07, 2009 at 15:30 UTC
This node was taken out by NodeReaper on Jul 07, 2009 at 15:30 UTC
Reason: ELISHEVA: re: - spam

You may view the original node and the consideration vote tally.

Re: On Parsing Perl
by quidity on Dec 04, 2000 at 04:40 UTC
I do most of my perl coding using CPerl mode for Xemacs, and although it is very good at spotting syntax it is often horribly wrong, especially when odd quoting characters or pod is brought into the equation. eval is even worse. I'd advise anyone even thinking of trying to parse perl to look at what can be achieved, and then either improving that (to the benefit of everyone) or just to give up.

I do sometimes find myself choosing a particular way of
coding over another (possibly better) way because the second breaks the pretty printing, and I want others using the same editor to be able to read the code I write.

**Re^2: On Parsing Perl**

by [Anonymous Monk](mailto:Anonymous.Monk@perl.com) on Aug 11, 2005 at 03:56 UTC

I hate to say this, but even as an XEmacs fan, cperl+gemacs is far superior. Recent GNU Emacs has some extra stuff that XEmacs doesn't have, that allows cperl to do some really amazing things. (I have vague memories of the "extra stuff" being multiple syntax transition tables for each character, so you can gracefully handle things like `m! x` and other non-standard delimiters. But I could be totally wrong.)

**Re: On Parsing Perl**

by [repson](mailto:repson@perl.com) on Dec 04, 2000 at 07:28 UTC

You could use [B::Deparse](http://search.cpan.org/perldoc?B::Deparse) for some of it, which eliminates some of what you don't want, but even that fails on many other things. The best bet is to code for the majority of perl and leave programmers to use their heads for the rest. This is what I do with syntax highlighting in vim, I use it generally but don't believe it for a moment. It is still sometimes helpful anyway. This is the way it will have to stay for now, at least until Perl6...

**Re: On Parsing Perl**

by [toadi](mailto:toadi@perl.com) on Dec 04, 2000 at 13:49 UTC

I'm with you [merlyn](mailto:merlyn@merlyn.com). I use vim(*nix) and textpad(windows), both make some mistakes. Like in some regex syntax it does some weird things...

--

My opinions may have changed, but not the fact that I am right

**Re: On Parsing Perl**

by [nop](mailto:nop@perl.com) on Dec 04, 2000 at 21:21 UTC

I use the perl mode on emacs, and resort to small tricks to keep everything ok. For example,

```
$'"/;
```

upsets the syntax colorization badly (as emacs thinks following code is in the string), so I use idioms like

```
$'"/; #"'
```

to "close" my "open" strings....
I just use backslashes for those. No need to "balance it".

~ Randal L. Schwartz, Perl hacker

I decided to run this test script under various versions of Perl.

```perl
@examples = split /
/, <<'EXAMPLES';
sin / ...
time / ...
localtime / ...
caller / ...
eof / ...
use constant FOO => 35; FOO / ...
use Fcntl qw(LOCK_SH); LOCK_SH / ...
sub no_args (); sub no_args{1}; no_args / ...
sub one_arg ($); sub one_arg{1}; one_arg / ...
sub normal (()); sub normal{1}; normal / ...
EXAMPLES
for (@examples) {
    s=/\n/..25 ; # /; die "this dies!";=
    local($a) = eval;
    $a = $@ if $@;
    print "$_
	$a
";
}
```

I don't know what the results would be for earlier versions, but from Perl 5.6 onwards it's pretty consistent.

<table>
<thead>
<tr>
<th></th>
<th>5.006</th>
<th>5.008</th>
<th>5.010</th>
</tr>
</thead>
<tbody>
<tr>
<td>sin</td>
<td>dies</td>
<td>dies</td>
<td>dies</td>
</tr>
<tr>
<td>time</td>
<td>49274891.72 49274891.74 49274891.76</td>
<td></td>
<td></td>
</tr>
<tr>
<td>localtime</td>
<td>dies</td>
<td>dies</td>
<td>dies</td>
</tr>
<tr>
<td>caller</td>
<td>dies</td>
<td>dies</td>
<td>dies</td>
</tr>
<tr>
<td>eof</td>
<td>dies</td>
<td>dies</td>
<td>dies</td>
</tr>
<tr>
<td>FOO</td>
<td>1.4</td>
<td>1.4</td>
<td>1.4</td>
</tr>
<tr>
<td>LOCK_SH</td>
<td>dies</td>
<td>dies</td>
<td>0.04</td>
</tr>
<tr>
<td>no_args</td>
<td>dies</td>
<td>dies</td>
<td>dies</td>
</tr>
<tr>
<td>one_arg</td>
<td>dies</td>
<td>dies</td>
<td>dies</td>
</tr>
<tr>
<td>normal</td>
<td>dies</td>
<td>dies</td>
<td>dies</td>
</tr>
</tbody>
</table>

Most of the "dies" instances also produced this message:

Warning: Use of "XXX" without parentheses is ambiguous at (eval N) line 1. However, the LOCK_SH example never generated errors, while the last three generated "Prototype mismatch" messages. I must...
also note that almost all of the examples generated warnings, despite using neither the '-w' option or 'use strict;'

Reaped: travesti
by NodeReaper on Nov 25, 2009 at 19:07 UTC
This node was taken out by the NodeReaper on Nov 25, 2009 at 19:34 UTC
Reason: kyle: reap spam
You may view the original node and the consideration vote tally.

Back to Meditations

PerlMonks lovingly hand-crafted by Tim Vroom
PerlMonks went on a couple dates, and then decided to shack up with The Perl Foundation
Wonderful Web Servers and Bandwidth Generously Provided by pair Networks
```perl
use v5.16.1;

sin / 25 ; # / ; die "this dies!" ;
```
use v5.16.1;

sin() / 25; # / ; die "this dies!";

sub BEGIN {
    require v5.16.1;
}
use strict;
no feature;
use feature ':5.16';

sin($_) / 25;
schw.pars.1 syntax OK
4. Operators

"Think of an operator as a special sort of function the
parser understands its operands as arguments."

Characteristics

Precedence: "governs when Perl should evaluate an
operator in an expression."

Associativity: "governs whether the operator
evaluates from left to right or
right to left.": exponentiation is right
associative 2 ** 3 ** 4 ...

Arity: number of operands

Fixity: an operators position relative to it's
operands $p++ (postfix)
also, infix, prefix, postfix, circumfix,
postcircumfix ( $p[3] ) ...

Types
Numeric, String, Logical, Bitwise, Special
these operators impose a context on their
operands, and thus it is important
to understand the values of the operands you
provide as well as the values
you expect to receive.

Special Operators:

auto-increment
repetition 'x'
infix range
which, in boolean context becomes: flip-flop
operator
3. The Perl Language