Language

A language is simply a set of valid sentences, or "what you can say". The concept of language presents two main problems: generation and recognition. This book is concerned with the latter, but to firmly understand the recognition of languages you need to understand generation as well. This chapter…

###Communication is not the exclusive domain of Man. Ants communicate with chemicals, birds define their domain with song, felines with scent, prairie dogs with calls, nerds with … ;) However, discounting the single-utterance grunts and calls, Man alone appears to possess the ability to communicate ideas through language [although after Douglas Adams' book I'm not so sure about the rats where we were all actually rats in their experiment…ultimate computing device or something???].

###How is it that a stream of grunts then can transmit information beyond lists of things? I.e., about time and space and that great turkey sandwich I just enjoyed?

###Humans are hardwired for grammar and speech. Who can forget the ease with which children pick up another language? All humans have speech and never simple caveman speech. In footnote, for a great debunking of the 200 words for snow in Eskimo language, see Pinker.

###something about how language changes over time.

###evolutionary pressure for language? Better able to pass on learned knowledge, making your offspring more robust. Chimps can show kids but can't write it down or tell them about it. Demonstration is required. Think about how a simple sentence can give you crucial information: friction gives off heat; e.g., rubbing two sticks together can make wood smoke.

###Who cares about grammars and machines and languages?

The Nature of Language

Cannot understand recognition until you understand the nature of languages; how they project meaning.

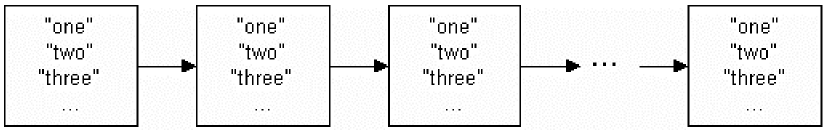
First show how to generate (finite) language, then infinite, then with some validity (probabilities), then show memory/dependencies "[…]", and finally ORDER requirements (nested […]).

Language Generation and Finite State Machines

Steven Pinker in his excellent book, *The Language Instinct*[1], provides a great example of a sentence generator: telephone directory assistance. You can build a simple, but unnatural sounding phone number voice synthesizer by recording a human saying each of the ten digits 0..9. To speak any telephone number, you pick the right recording at the right digit position. For example, to say 555-1212, your program would say the recording for "five" three times then

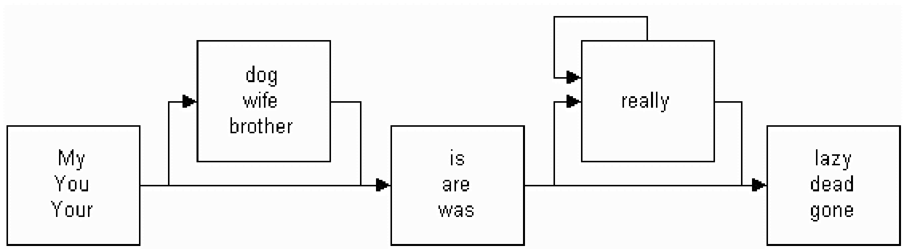
[1] HarperPerennial, 1994. Pinker's book has greatly influenced my thinking about languages and structure. My understanding of how humans generate language is rooted in his work. The state diagrams shown in this section are inspired by the machines in *The Language Instinct*.

"one", then "two", etc… This problem screams for a state machine solution like the following machine with seven states:



This machine defines a large, but finite set of sentences because there are exactly 7^10 possible combinations of digits.

Most languages are not finite, however. When I was a suffering undergraduate student at Purdue University (back before GUIs), I ran across a sophisticated sentence generator, a "documentation generator", that automatically produced verbose formal manuals. You could read about half a paragraph before your mind said, "Whoa! That doesn't make sense." Still it was amazing that a program could produce a document that, at first glance, was human-generated or at least written by a manager masquerading as a coder. How can a program generate English sentences? Such a program would also use some form of state machine, but with more states and more choices per state than the telephone number voice synthesizer. For example, here is a partial state machine for picking Blues song lyrics:



The set of sentences generated by this state machine is not finite because you can always follow the loop around "really" an infinite number of times. The notion that you can generate an infinite language with a finite generator or description is a crucial concept that you will see again.

This state machine can produce lyrics such as "My dog is really lazy" and "You was gone", but also bogus statements such as "My Brother are dead"[2]. Rather than choose words at random at each state, the machine can use known probabilities for how often certain words follow each other. That scheme helps, but no matter how good your statistics are, the machine will eventually generate an invalid sentence. Surely this is a poor cousin to the mechanism used in the human mind for language generation.

Consider language generation from the perspective of the adventurer in the maze ### whereas the last chapter discussed the maze metaphor from the recognition point-of-view. The language defined by the maze is the set of all passphrases and, hence, you can generate all possible sentences by walking every possible path from the entrance

[2] What happens if you run this Blues lyrics machine backwards? The old joke says, "you get your dog back, your wife back, …"

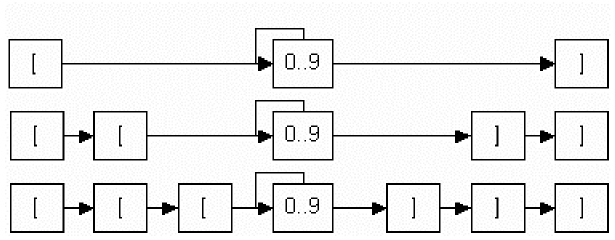
to the exit.

Imagine that at least one loopback exists in paths of the maze. You could walk around forever, making an infinitely-long passphrase. The maze can therefore simulate a finite or infinite language generator and is a metaphor for a state machine.

Think about why state machines can generate invalid sentences. There are three reasons[3]:

1. Grammatical does not imply sensible (and vice versa). For example, "cows flow supremely" is grammatically okay, but makes no sense. In English, this is self-evident. On the other hand, rock-n-roll lyrics and poetry, when read literally, often make no sense. In a computer program, you also know that a syntactically valid assignment such as "`employeeName=milesPerGallon;`" can make no sense depending on what your program is meant to do. Unless a human is doing the generation or recognition, nonsensical sentences are an issue (at least until Isaac Asimov's vision of robotics becomes a reality).

2. There are dependencies between the words of a sentence. If confronted with a ']' (close square bracket), there is not a programmer in the World that would not have an involuntary response to look for the opening '['. ###add knock knock joke? You could design a state machine that had paths to generate one pair of brackets, two pair of brackets, three pair of brackets ad naseum, but such a machine would be infinitely large. Here is a machine that generates from one to three brackets around an integer.



3. There are order requirements between the words of a sentence. When a program has an array reference nested inside a parenthesized expression, you expect the closing square bracket and parenthesis to be in a particular order in relation to each other as opposed to the opening bracket or parenthesis, which the previous rule dictates. For example, you immediately see "(a[i+3)]" as invalid. A state machine has no way to encode or remember what order it saw previous symbols.

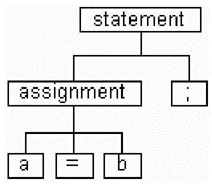
In order to design a machine capable of solving problems 2 and 3, think about the difference between a list of words ala the state machines and what really dictates the validity of sentences. The key idea is that a sentence is not a cleverly combined sequence of words, but rather groups of words and groups of groups. In other words, sentences have *structure* like this book. This book is organized into a series of chapters each containing sections, which themselves contain subsections and so on. Nested structures abound in computer science too. For example, in an object-oriented class library, classes group all elements beneath them in the hierarchy into a categories (any kind of cat might be a subclass of Feline etc…). The first hint of a solution to our

[3] *The Language Instinct*, pp 93-97. I use Pinker's reasons, but in a computer language context.

underpowered state machines is now apparent. Just as a class library is not a flat list of classes, a sentence is not just a flat list of words.

Language Generation and Pushdown Machines

The structure of Human and most computer language sentence structure is described by a tree, a two-dimensional framework with words at the leaves and grouping nodes as subtree roots. For example, the assignment "`a=b;`" can be represented by the following tree.



The non-leaf nodes, "statement" and "assignment", encode the underlying structure of the four symbols 'a', '=', 'b', and ';'.[4]

In your head, language generation does not begin with the first word of a sentence. You generate language at an abstract level and work your way down, albeit very quickly. When you write code, you do not think like a state machine: "I need a variable then an equals etc…". You think: "I need to assign 'b' to 'a'". Then you worry about the details of the assignment. This smacks of the top-down recognizers from the previous chapter.

So walking the states of a state machine is too simple an approach for the generation of complex language. There are word dependencies and order requirements among the output words that it cannot satisfy, as the state machine has no memory of what it generated in the past. To design a machine that generates language by structure rather than word sequence, you need a memory system that can pair up dependent words and can enforce nested structure.

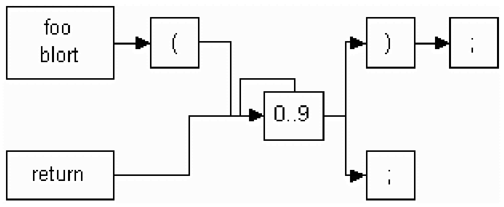
It turns out that the humble stack is the perfect memory structure to solve both word dependency and order problems, which is great because modern computers[5] have hardware stacks. Yep, adding a method call and return scheme to the state machine turns it into a sophisticated language generator. You call such a device a *pushdown machine*.

[4] You might recall being tortured in elementary school by so-called "sentence diagramming" exercises where you had to figure out how the words modified each other. Sentence structure trees are akin to these "fishbone" diagrams. I say "elementary school" rather than "grammar school", as Americans often refer to that early period of bondage, to avoid any confusion that Americans were learning grammatical structure in school. I dare say that we learn about grammars only when studying a foreign language later in life.

[5] The idea of pushing the return address onto a stack before branching to implement a function call (so that you can continue where you left off upon return) is so commonplace now that we forget that many computers in the 1960s did not have a hardware stack. For example, the Control Data Corporation's 6000 series computers did not have a stack (60 bit words, ones-complement arithmetic versus twos-complement, one (non-floating) instruction called "set", etc… These were Seymour Cray's early babies if memory serves me correctly). You could store a single return address in the first word of the invoked function, which the compiler left open for you. Fortran in those days could not do recursion. As my friend Gary Funck says, "if you're young enough that you do not remember CDC, then good for you."

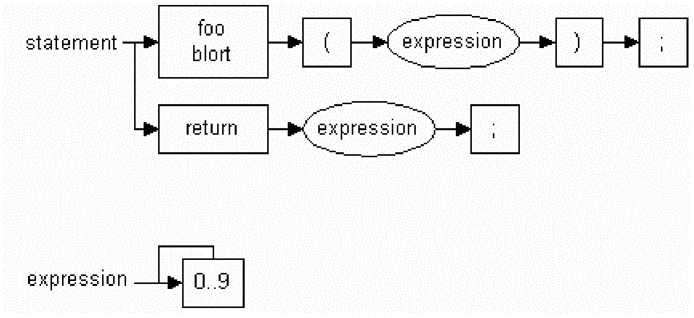
Solving Word Dependencies

A simple pushdown statement generator demonstrates how a stack lets you enforce word dependencies. Consider a language with method call statements, return statements, and integers for expressions. The appearance of a closing parenthesis depends on the existence of a previous, opening parenthesis. The state machine component of the pushdown machine is a good place to start:

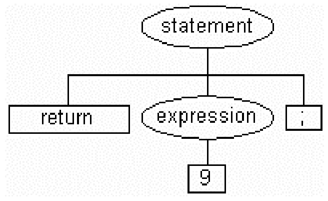


You can generate "`foo(34);`", "`blort(193);`", and "`return 9;`". Unfortunately, you can also generate the invalid sentences "`foo(1;`" and "`return 5);`". The problem is that the digit loop state cannot remember whether the method call or return statement transitioned to it, hence, it does not know to which of the emanating states to jump. The machine cannot encode the dependency between the opening and closing parenthesis.

Adding a stack allows you to store a "return address", which lets you partition a state machine into smaller machines invoked like normal procedures. Splitting off the digit loop state and labeling the two submachines results in the following pushdown machine:



To distinguish between nodes that generate a word directly and invocations of submachines, use rectangular states for generators and oval states for submachine invocation states. Pushdown machines specified graphically in this manner are called *syntax diagrams*. The labels of the submachines correspond to the grouping constructs found in a sentence's tree structure representation. The sentence structure of "`return 9;`" in tree form looks like:



A sentence's tree structure, which you will later call a *derivation tree* or *parse tree*, is a record of how a particular pushdown machine generated that sentence. The easiest way to think about these trees is that they are a record of the call sequence among the submachines. In this case, submachine "statement" generated "`return`" then called submachine "expression" to generate "`9`" and then "statement" generated the "`;`".

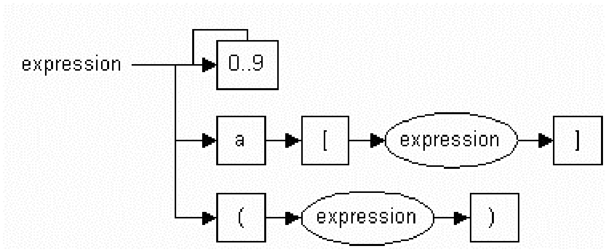
If you wanted to, you could build such generator with Java methods as follows.

```
void statement() {
  System.out.println("return");
  expression();
  System.out.println(";");
}
void expression() {
  // generate a sequence of digits
  while ( some-condition ) {
    System.out.print(some-digit);
  }
}
```

The call tree for generating "`return 9;`" corresponds one-to-one to the generated sentence structure tree.

Solving Word Order Requirements

A language with word order requirements normally has nested substructures. Reconsider the problem of pairing up square brackets and parentheses in the right order. For example, you must close a bracketed subexpression before closing an outer parenthesized expression such as "`(a[34])`". Here is a pushdown machine that generates "`(a[34])`" as well as others.



This expression generator can generate expressions like "`29342`", "`a[12]`", "`(89)`", "`a[(1)]`", and "`(a[a[1]])`". The order of word generation within any submachine is guaranteed even if it invokes another submachine (or itself in this case).[6] For the second alternative, "`a[`*`expression`*`]`", the '`]`' is guaranteed to occur after the index expression. Any structure generated by the index expression will terminate before the '`]`'. Similarly, the third alternative guarantees that the '`)`' will occur after the structure generated by the enclosed expression. That is, the machine ensures that grouping symbols are appropriately nested. Improper nesting is abhorrent to most minds; imagine the start of a chapter before the end of its preceding chapter.

To summarize, sentences are not cleverly chosen lists of words, but groups of words and groups of groups structured as a tree with the words at the leaves and group names as subtree roots. A finite state machine can generate simple finite or infinite sentences adequately like the directory assistance synthesizer, but state machines cannot generate sentences that have word dependencies

[6] Do not let the recursion scare you. Languages are naturally highly recursive beasts. As L. Peter Deutsch said, "To iterate is human, to recurse divine." I see another play on the same phrase in people's email signature: "To error is human, to moo bovine."

or nested substructures like word order requirements. A pushdown machine is a state machine augmented with a stack that allows you to break up the machine into labeled procedure-like submachines and makes a pushdown machine much more powerful than a state machine. A sentence's tree structure is a record of the submachine invocations used to generate that sentence.

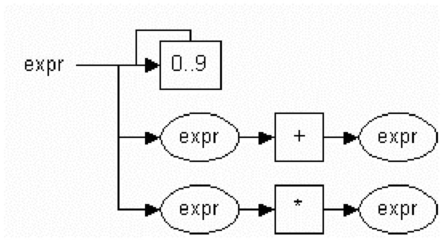
Ambiguous Language Generators (###Language or Generator?)

The Language Instinct cites a marvelously ambiguous statement by Groucho Marx: "I once shot an elephant in my pajamas. How he got into my pajamas I'll never know." As the reader, you can decide that either Groucho shot an elephant sporting (and presumably stretching) his pajamas or Groucho was wearing his pajamas and the elephant happened to be lurking inside the same pajamas.

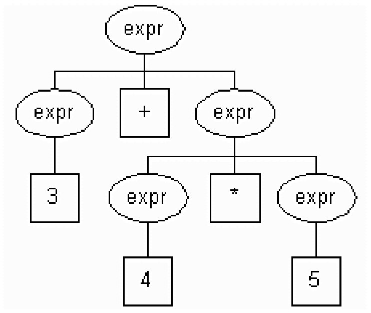
Ambiguity is a source of humor in English, but the bane of computing. Computers must always know exactly how to interpret every statement or, at the lowest level, computers must always make decisions *deterministically*—they must know exactly which path to take. A classic example of an ambiguous statement relates to the generation of expressions containing operators. The expression 3+4*5 is not ambiguous to an adult human. It means multiply four times five and add three yielding 23. An elementary school student doing the operations from left to right might ask, "Why is the result not 35?" Indeed, why not? Because mathematicians have decreed it so. In fact, German mathematician Leopold Kronecker went so far as to say, "God made the natural numbers; all else is the work of man." The following pushdown machine generates expressions with operators plus and multiply.

###some languages are inherently ambiguous.

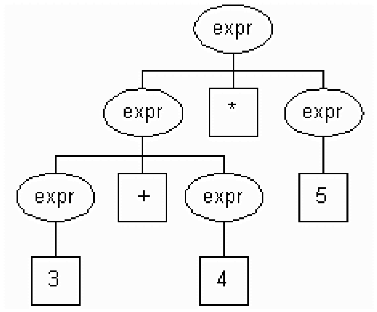
###is left-recursive stuff really easy to understand?



An adult generates 3+4*5 by recursively calling submachine expr until they arrive at the following interpretation of the inherent structure:



The elementary school child would ask why you do not generate 3+4*5 with the following structure, grouping the 3 and 4 together instead of the 4 and 5.



A machine, state or pushdown, that can generate a sentence in more than one way is an ambiguous language generator because it can mean two different things when generating the same sentence. In other words, if a machine can yield more than one tree structure for a given sentence, that machine is ambiguous as the sentence structure is ambiguous. Ambiguity is a problem because structure is the basis of meaning and leads to problems regardless of whether you are generating or recognizing sentences.

Grammars

A language can be an infinitely large set of sentences, making it impossible to define languages by simply delineating the sentences. As you saw above, finite state machines and finite pushdown machines can generate infinite sets of sentences and, in fact, are a way of formally defining a language. By walking all possible paths of a machine, you generate all possible sentences in the machine's language.

Machine diagrams are easy to understand due to their visual nature, but diagrams are tiresome to build and very hard for a computer to understand. Further, diagrams are almost free form in that two people could build two different diagrams for the same machine design. We need a better means of formally describing languages.

Abstacting Structure From Exemplars

Before blurting out the answer to our language definition woes, see if you can analyze a few English sentences to abstract their underlying structure. Proficiency at determining language structure from representative sentences must become second nature to you. Look at the following exemplar sentences that define a subset of English.

```
John gesticulates
John gesticulates vigorously
The dog ate steak
The dog ate ravenously
```

The sentences have meaning to you because your brain is subconsciously associating concepts with the words and structuring the words into phrases and groups of phrases, which conveys meaning. Try to do this consciously by abstracting structure from these exemplars. There is always a person or thing (a *subject*) and a verb describing an action (a *verb phrase*) and sometimes an *object* that the subject acts upon. Replace the phrases in the sentences of similar structure with a symbol representing that structure. First, abstract all the subjects:

```
Subject gesticulates
Subject gesticulates vigorously
Subject ate steak
Subject ate ravenously
```

Next, abstract the verb phrases:

```
Subject VerbPhrase
Subject VerbPhrase
Subject VerbPhrase steak
Subject VerbPhrase
```

Finally, abstract the objects:

```
Subject VerbPhrase
Subject VerbPhrase
Subject VerbPhrase Object
Subject VerbPhrase
```

Removing duplicate abstract sentences leaves only two types of sentences and reveals the overall structure of a sentence:

```
Subject VerbPhrase
Subject VerbPhrase Object
```

Turning to the substructures, the subject phrases:

```
John
The dog
```

are further structured as:

```
Noun
Determiner Noun
```

The verb phrases:

```
gesticulates
gesticulates vigorously
ate
ate ravenously
```

are structured as:

```
Verb
Verb Adverb
```

You can summarize the finite language defined by the exemplars with the following set of syntax rules, or *grammar*.

1. A sentence is a subject followed by a verb phrase optionally followed by an object.
2. A subject is a noun optionally preceded by a determiner.
3. A verb phrase is a verb optionally followed by an adverb.
4. A noun is `John` or `dog`.
5. A verb is `gesticulates` or `ate`.
6. An adverb is `vigorously` or `ravenously`.
7. An object is `steak`.
8. A determiner is `The`.

The structure of these rules is right, given our intuitive understanding of English, but the rules are actually too loose because they generate more than the four sentences in our language. For example, the rules let you say, "`dog gesticulates ravenously`". You will encounter this situation frequently as you build grammars. There is usually a tradeoff between writing a natural description and achieving strict conformance.

Syntax Versus Semantics

The fact that these rules for a subset of English are loose also highlights that a language can be hard to describe with a series of simple syntactic rules. There are semantic rules (rules about which sentences are meaningful) in addition to the syntactic rules (rules in the grammar). The rules above are strictly syntactic and, at least for computer languages, are the easiest to specify.

To see the difference between syntactic and semantic rules, reconsider the arithmetic expressions and the associated ambiguous machine generator described above. You can specify the same language as the pushdown machine with the following recursive rule.

1. An expression is an integer or two smaller expressions separated by a '+' or two expressions separated by a '*'.

The machine and equivalent rule are ambiguous because they are not complete specifications for the arithmetic language understood by human adults. Given another rule, a semantic rule, your specification becomes unambiguous:

2. Operator '*' has higher precedence than operator '+'.

This rule, decreed by mathematicians, forces a unique tree structure for any given expression (the '*' operator binds more tightly than '+'). The problem with this rule, and the reason the pushdown machine did not encode this rule, is that encoding semantics rules is often hard or expensive for a computer. ### Ref to sem preds and context-free sections?

Human language has so many special cases and semantic rules that defining a precise grammar is almost impossible. ###Esperanto. I once saw a book on English grammar (###lookup at amazon?) that was about 4 inches thick. Besides English is a moving target as is any human language currently in use. Consider the difference between Elizabethan English and Modern English.

###sometimes you can get the semantics right via syntax (normally requiring non-CFG), but expression thing can be rewritten.

Bear in mind as you discover meta-languages in the next section that there is sometimes a gap between the language you mean and the language you can specify easily.

Language Definition Meta-Languages

There is another problem with the rules as described thus far: English is neither precise nor terse, making English a lousy language for describing languages (a lousy *meta-language*). Just as languages themselves evolve over time, the manner in which computer scientists have specified languages has evolved.

###This section defines three such…

###CFG notation is used for theory discussions in advanced topic section.

###what do they have in common?

###follow chrono sequence of meta-language definition

Context-Free Grammar Notation

[*Skip this section if you do not plan on reading the Advanced Topics section*]

The first meta-language used extensively and the meta-language preferred by computer language theorists is called *CFG* (context-free grammar) notation (there are a number of dialects). CFG specifications provide a list of rules with left- and right-hand sides separated by a right-arrow symbol. One of the rules is identified as the *start rule* or *start symbol*, implying that the overall structure of any sentence in the language is described by that rule. The left-hand side specifies the name of the substructure you are defining and the right hand side specifies the actual structure (sometimes called a *production*): a sequence of references to other rules and/or words in the vocabulary. The rules are of the form:

Rulename → finite sequence of rule references and token references

For convenience, rule names (*nonterminals*) are always single uppercase letters and vocabulary words, *terminals*, are always single lowercase letters. CFG grammars are presumed to specify the syntactic structure not the lexical structure so terminals are token types really that represent a class of input symbols. For example, you could decree that rule *I* means "integer" and that token *d* represents any digit. Consequently, to specify that an integer can be any single digit, you would write:

$I \rightarrow d$

You could also have used a higher-level symbol and said that token *i* represents any integer.

Non-character literal input symbols are allowed as long as they are one character (such as the '+' sign). If a structure name can be one of several alternatives, the structure name is repeated on the left-hand side. The follow grammar indicates that rule *A* can be either *a* or *b*.

$A \rightarrow a$
$A \rightarrow b$

The right-hand side may be empty:

$A \rightarrow$

Empty productions are useful for saying that a construct is optional. For example, the following grammar generates the language *a* or *ab*.

$A \rightarrow aB$
$B \rightarrow b$
$B \rightarrow$

Sometimes you will see empty productions written:

$A \rightarrow \varepsilon$

The ε is used as a placeholder sort of like the IBM manuals that say "this page left intentionally blank" and is the reason that empty productions are often called *epsilon productions*.

Using CFG notation, it is easy to describe an arithmetic expression language with plus and multiply operators as generated by the pushdown machine seen above. In the following grammar, *E* is the start rule.

$E \rightarrow i$
$E \rightarrow E+E$
$E \rightarrow E*E$

The first rule, *E* for "expression", defines three alternative substructures or *productions* as they are called. The first production indicates that an expression can be an integer. The second and third productions say that an expression can be two smaller expressions separated by a '+' or a '*' sign, respectively.

How can you generate "*i+i*i*" from this grammar? Start with the second alternative of *E*, *E+E*, and then replace the second reference to *E* by *E*E*, yielding *E+E*E*. Finally, replace all references to *E* with *i*. Recall that the pushdown machine and, hence this grammar, are ambiguous because the grammar can also generate "*i+i*i*" in a different way (begin with rule three not rule two). In summary, the difference looks like this:

1. $E \Rightarrow E+E \Rightarrow E+E*E \Rightarrow i+i*i$

2. $E \Rightarrow E*E \Rightarrow E+E*E \Rightarrow i+i*i$

Regular Expression Notation

Classically, CFG used for syntax and regular expression notation used for lexical languages.

"match" or "generate" in this section?

CFG notation is great for describing pushdown machines, but is overkill for describing finite machines because finite machines do not have submachines that you can call; right-hand sides cannot reference other rules. Instead, programmers normally use *regular expressions* to specify finite machines. Oddly, the regular expression syntax is more complicated, but only because, without recursion (no stack), you cannot specify repeated structures. A regular expression does need a rule name, but when used in translator generators, they are written as rules with rule names to identify the myriad of expressions describing vocabulary symbols. An expression contains single characters and possibly constructs to specify looping and so on. The following rule has a regular expression right-hand side and generates a single input sequence, "abc":

```
simple : abc ;
```

The ':' separates the left- and right-hand sides and the ';' terminates the rule. Alternative substructures are separated with '|'. The following rule matches either "abc" or "duh".

```
simple : abc
       | duh
       ;
```

You will also see character sets such as `[a-z0-9]`, which means any lowercase letter or any digit.

Regular expressions also have constructs to modify the number of times an element can occur:

Construct	Description	Example
`expression`?	The expression is optional.	`ab?c` means `abc` or `ac`. `a(b|z)?c` means `abc`, `azc` or `ac`.

*expression**	The expression can occur zero or more times. This is called the *closure* of the expression.	`[a-zA-Z][a-zA-Z0-9]` describes any variable name consisting of letters or digits as long as it begins with a letter.
expression+	The expression can occur one or more times. This is called the *positive closure* of the expression.	`(0|1)*` means a binary number.

BNF Notation

Language theorists love CFG notation, but most language reference guides use BNF (Backus-Naur Form) notation, which is really just a more readable version of CFG notation. All rule names are surrounded by <…> and → is replaced with "::=". Also, alternative productions are separated by '|' rather than repeating the rule name on the left-hand side. BNF is more verbose, but has the advantage that you can write meaningful rule names and you are not constrained vis-à-vis capitalization. Rules are therefore of the form:

```
<rulename> ::= production 1
           |   production 2
           …
           |   production n
```

The eight rules for the subset of English described previously are written in full English, but you can say the same thing more tersely as follows using BNF notation:

```
<Sentence>   ::= <Subject> <VerbPhrase> <Object>
<Subject>    ::= <Determiner> <Noun>
<VerbPhrase> ::= <Verb> <Adverb>
<Noun>       ::= John | dog
<Verb>       ::= gesticulates | ate
<Adverb>     ::= vigorously | ravenously |
<Object>     ::= steak |
<Determiner> ::= The |
```

dffd

YACC Notation

YACC[### ref] introduced a simpler derivative without <…> at cost of upper/lower case indicator for terminal/nonterminal. YACC notation is important as it was the de facto standard for around 20 years; YACC notation is the progenitor for ANTLR notation. It is a more terse version of BNF.

```
sentence   : subject verbPhrase object ;
subject    : determiner noun;
verbPhrase : verb adverb ;
noun       : "John"
           | "dog"
           ;
verb       : "gesticulates"
           | "ate"
           ;
adverb     : "vigorously"
```

```
            | "ravenously"
            |
            ;
object      : "steak"
            |
            ;
determiner  : "The"
            |
            ;
```

ANTLR Notation

ANTLR uses YACC notation with EBNF constructs, borrowed from regular expression notation

```
sentence   : subject verbPhrase (object)? ;
subject    : (determiner)? noun;
verbPhrase : verb (adverb)? ;
noun       : "John"
           | "dog"
           ;
verb       : "gesticulates"
           | "ate"
           ;
adverb     : "vigorously"
           | "ravenously"
           ;
object     : "steak"
           ;
determiner : "The"
           ;
```

Make easier using categories of words.

```
sentence   : subject verbPhrase (object)? ;
subject    : (DETERMINER)? NOUN;
verbPhrase : VERB (ADVERB)? ;
object     : NOUN ;
```

But, this is even looser.

Talk about using char as elements to show same syntax. No reason limited to token types.

describe looping / grouping

categories: group instances of ids into ID. don't have choices necessarily at each node.

How to group things…left/right associative…

Grammars, interestingly enough, conforms to a language (a meta-language) in its own. ANTLR notation in its own words:###.

for any language, infinite # of CFGs.

Regular Grammars

Map finite machine to grammar and you have a special property that right-hand sides are composed exclusively of vocabulary symbols or at most a single rule reference at the extreme right edge. (check the last part###).

Can use for char themselves.

People often use rexprs.

Here is where I start to unify things.

Tree Grammars

This should raise a few eyebrows. How can you write a grammar for a two-dimensional structure like a tree? Easy. Add notation to indicate the tree structure ala LISP.

```
expr : #( PLUS expr expr )
     | INT
     ;
```

Why would you want to specify a two-dimensional language? Intermediate forms.###

Derivations

Now that they you are familiar with languages and grammars that describe them, …

relate to walk of machine diagram.

$\Rightarrow\varepsilon$

apply production/rule

order of derivation not specified in tree? same tree structure associated with multiple derivations.

leftmost rightmost derivations: when more than one nonterminal in a sentential form, specifies which to pick.

Also the case as you saw above, there may be more than a single tree for a single string that has multiple derivations. That is, different derivations can lead to the same tree if only a order of substitution issue and diff deriv can lead to diff trees. 1 string has 1 or more trees and each tree has 1 or more derivations. each tree, there is a unique rightmost and leftmost deriv.

parse trees

Structure Imparts Meaning

This highlights very clearly that structure imparts meaning!

Give sentence with clear meaning and then add commas to change subject/meaning.

Terence says the emperor has no clothes

versus

Terence, says the emperor, has no clothes

##You saw earlier that grammatical does not imply sensible. It is also the case that the way you group things projects meaning for sure.

Ambiguities; else-ambiguity (sometimes need more than grammar) unfortunately necessary: most languages have pushdown machine description and set of additional constraints; use C++ example of expression vs declaration. Find slide in talks.

Different kinds of languages (Type 0..4)

that also highlights the gap between structure and meaning

Talk about them in context of grammars; show what the grammar constraints are.

Regular Languages (Type 4)

Content-Free Languages (Type 3)

Context-Sensitive Languages (Type 2)

variable decl must precede use. Can't do so we use ID placeholder and augment parser with symbol table memory. Or use BASIC for loop and next VAR statement as context.

Remember the gap described above between what you want to describe and what you can? Well, CFG are a subset of grammars.

Grey area between what is syntax and what is semantics sometimes (like the variable use before def thing).

Languages (Type 0)

What to do in the syntax versus semantics (like (…)7 vs ({7}? …)*