

The Daily PL - 6/13/2023



So far in this course we have covered lots of material. We've learned lots of definitions, explored lots of concepts and programmed in languages that we've never seen before. In all that time, though, we never really got to the bottom of one thing: *What exactly is a language?* In this module, we are going to cover exactly that!

Back to Basics

Let's think way back to the beginning of the semester and recall two very important definitions: syntax and semantics. On a **day long ago** (<https://uc.instructure.com/courses/1610326/pages/the-daily-pl-5-slash-10-slash-2022>) we defined *semantics* as the effect of each statement's execution on the program's operation and we defined *syntax* as the rules for constructing structurally valid (note: valid, not correct) computer programs in a particular language. There's that word again -- *language*.

Before starting to define *language*, let me offer you two alternate definitions for syntax and semantics that draw out the distinction between the two:

- The syntax of a programming language specifies the *form* of its expressions, statements and program units.
- The semantics of a programming language specifies the *meaning* of its expressions, statements and program units.

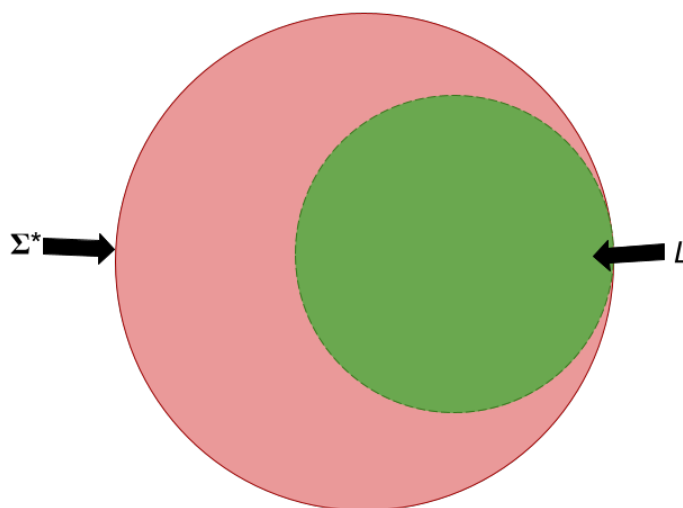
It is interesting to see those two definitions side-by-side and realize that they are basically identical with the exception of a single word! One final note about the connection between syntax and semantics before moving forward: Remember how a well-designed programming language has a syntax that is evocative of meaning? In other words, a **well-designed language**  (<https://stackoverflow.com/questions/1345843/what-does-the-question-mark-at-the-end-of-a-method-name-mean-in-ruby>) might allow variables to contain a symbol like  which would allow the programmer to indicate that it holds a Boolean.

Before we can specify a syntax for a programming language, we need to specify the language itself. In order to do that, we need to start by defining the language's *alphabet* -- the finite set of characters that can be used to write sentences in that language. We usually denote the alphabet of a language with the Σ . It is sometimes helpful to denote the set of all the possible sentences that can be written using the characters in the alphabet. We usually denote that with Σ^* . For example, say that $\Sigma = \{a, b\}$, then $\Sigma^* = \{a, b, aa, ab, ba, aaa, aab, aba, abb, \dots\}$. Notice that even though $|\Sigma|$ is finite (that is, the number of elements in Σ is finite), $|\Sigma^*| = \infty$.

The alphabet of the C++ programming language is large, but it's not infinite. However, the set of

sentences that can be written using that alphabet is infinite. But, as we all learn early in our programming career, just because you can write out a program using the valid symbols in the C++ alphabet does not mean the program is syntactically valid. The very job of the compiler is to distinguish between valid and invalid programs, right?

Let's call the language that we are defining L and say that its alphabet is Σ . L can be thought of as the set of all valid sentences in the language. Therefore, every sentence that is in L is in Σ^* -- $L \subseteq \Sigma^*$.



Look closely at the relationship between Σ^* and L . While L never includes a sentence that is not included in Σ^* , they can be identical! Think of some languages where any combination of characters from its alphabet are valid sentences! Can you come up with one?

The Really Bad Programmer

So, how can we determine whether a sentence made up of characters from the alphabet is in the language or not? We have to be able to answer this question -- one of the fundamental jobs of the compiler, after all, is to do exactly that. Why not just write down every single valid sentence in a giant chart and compare the program with that list? If the program is in the list, then it's a valid program! That's easy.

Not so fast. Aren't there an infinite number of valid C++ programs?

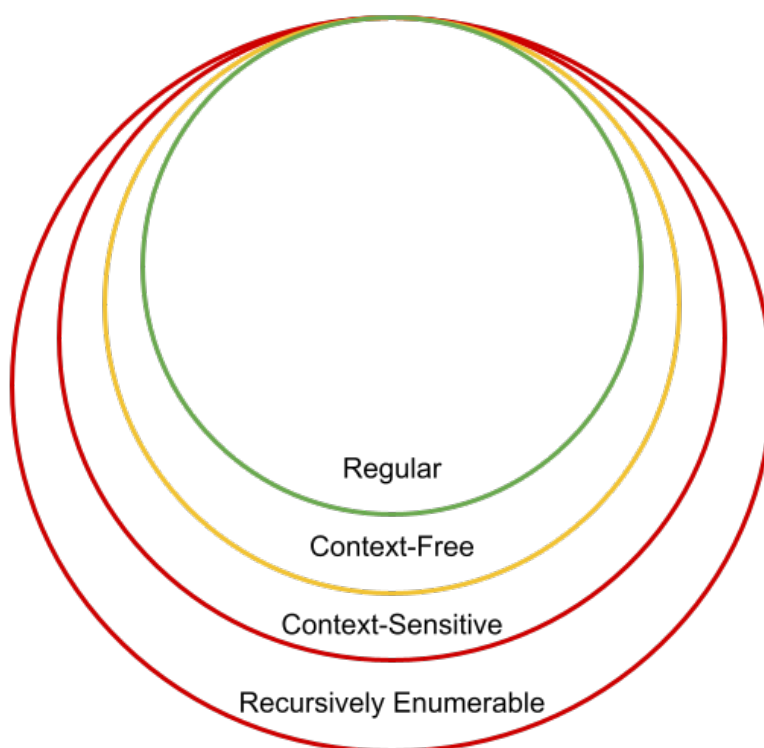
```
int main() {  
    if (true) {  
        if (true) {  
            if (true) {  
                ...  
                std::cout << "Hello, World.";  
            }  
        }  
    }  
}
```

Well, dang. There goes that idea.

It would be great to have a tool for languages that allows us to create something that *recognizes* and *generates* valid sentences in the language. We will do exactly that!

Language Classes

The famous linguist Noam Chomsky was the first to recognize how there is a hierarchy of languages. The hierarchy is founded upon the concept of how easy/hard it is to write a concise definition for the language's valid sentences.



Each level of Chomsky's Hierarchy, as it is called, contains the levels that come before it. Which languages belong to which level of the hierarchy is something that you will cover more in CS4040. (Note: The languages that belong to the *Regular* level can be specified by **regular expressions** https://en.wikipedia.org/wiki/Regular_expression). Something I know some of you have written before!)

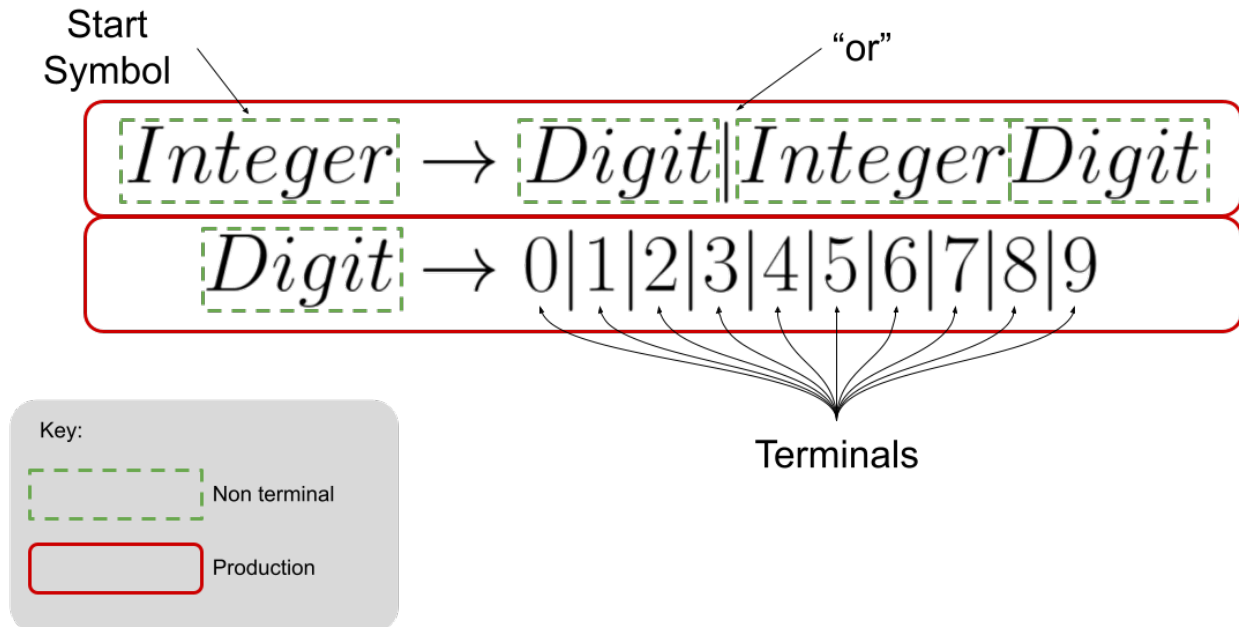
For our purposes, we are going to be concerned with those languages that belong to the Context-Free level of the hierarchy.

Context-Free Grammars

The tool that we can use to concisely specify a Context-Free language is called a *Context-Free Grammar*. Precisely, a Context-Free Grammar, G , is a set of productions P , a set of terminal symbols, T , and a set of non-terminal symbols, N , one of which is named S and is known as the

start symbol.

That's a ton to take in. The fact of the matter, however, is that the vocabulary is intuitive once you see and use a grammar. So, let's do that. We will define a grammar for listing/recognizing all the positive integers:



Now that we see an example of a grammar and its parts and pieces, let's "read" it to help us understand what it means. Every valid integer in the language can be *derived* by writing down the start symbol and iteratively replacing every non-terminal according to a production until there are only terminals remaining! Again, that sounds complicated, but it's really not. Let's look at the derivation for the integer 142:

$\textit{Integer} \rightarrow \textit{IntegerDigit}$
 $\rightarrow \textit{IntegerDigitDigit}$
 $\rightarrow \textit{DigitDigitDigit}$
 $\rightarrow 1\textit{DigitDigit}$
 $\rightarrow 14\textit{Digit}$
 $\rightarrow 142$

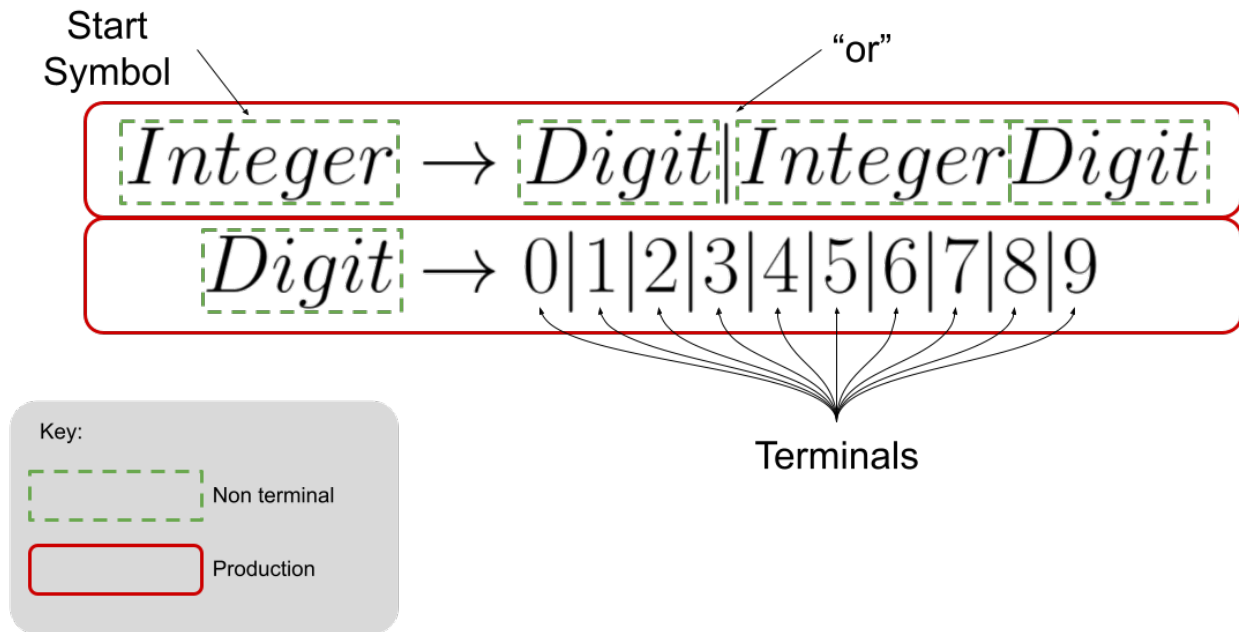
We start with the start symbol *Integer* and use the first production to replace it with one of the two options in the production. We choose to use the second part of the production and replace *Integer* with *Integer Digit*. Next, we use a production to replace *Integer* with *Integer Digit* again. At this point we have *Integer Digit Digit*. Next, we use one of the productions to replace *Integer* with *Digit* and we are left with *Digit Digit Digit*. Our next move is to replace the left-most *Digit* non-terminal with a terminal -- the 1. We are left with 1 *Digit Digit*. Pressing forward, we replace the left-most *Digit* with 4 and we come to 14*Digit*. Almost there. Let's replace *Digit* with 2 and we get 142. Because 142 contains only terminals, we have completed the derivation. Because we can *derive* 142 from the grammar, we can confidently say that 142 is in the language described by the grammar. You can say that a sentence is in the language, *by construction*, if you can construct a derivation for that sentence using the productions of the grammar.

(Context-free) Grammars (CFGs) are a mathematical description of a language-generation mechanism. Every grammar defines a language. The strings in that language are those that can be derived from the grammar's start symbol. Put another way, a CFG describes a process by which all the sentences in a language can be enumerated, or generated.

Defining a way to generate all the sentences in a grammar is one way to specify a language. Another way to define a language is to build a tool that separates sentences that are in the language from sentences that are not in the language. This tool is known as a recognizer. There is a relationship between recognizers and generators and we will explore that in the future.

Grammatical Errors

Previously we worked with a grammar that purported to describe all the strings that looked like integers:



We performed a derivation using its productions to prove (*by construction*) that 142 is in the language. But, let's consider this derivation:

$$\begin{aligned}
 Integer &\rightarrow Integer Digit \\
 &\rightarrow Integer Digit Digit \\
 &\rightarrow Digit Digit Digit \\
 &\rightarrow 0 Digit Digit \\
 &\rightarrow 01 Digit \\
 &\rightarrow 012
 \end{aligned}$$

We have just proven (again, by construction) that 012 is an integer. This seems funny. A number

that starts with 0 (that is not just a 0) should *not* be deemed an integer. Let's see how we can fix this problem.

In order to guarantee that we cannot derive a number that starts with a 0, we will need to add a few more productions. First, let's reconsider the production for the start symbol Integer. At the very highest level, we know that 0 is an integer. So, our start symbol's production should handle that case.

With the base case accounted for, we next consider that an integer greater than 0 cannot start with a zero, but it can start with any other digit between 1 and 9. It seems handy to have a production for a non-terminal that expands to the terminals 1 through 9. We will call that non-terminal *Nzd* for *non-zero digits*. After the initial non-zero digit, an integer can contain zero or more digits between 0 and 9 (we can call this the *rest* of the integer). For brevity, we probably want a production that will expand to all the digits between 0 and 9. Let's call that non-terminal *Zd* for zero digits. We'll define it either as a 0 or anything in *Nzd*. If we put all of this together, we get the following grammar:

$$\textit{Integer} \rightarrow \textit{Zd} | \textit{Nzd} \textit{Rest}$$

$$\textit{Rest} \rightarrow \textit{Zd} | \textit{Zd} \textit{Rest}$$

$$\textit{Zd} \rightarrow 0 | \textit{Nzd}$$

$$\textit{Nzd} \rightarrow 1 \dots 9$$

Let's look at a few derivations to see if this gets the job done. Is 0 an integer?

$$\begin{aligned} \textit{Integer} &\rightarrow \textit{Zd} \\ &\rightarrow 0 \end{aligned}$$

And a single-digit integer?

$$\begin{aligned} \textit{Integer} &\rightarrow \textit{Zd} \\ &\rightarrow \textit{Nzd} \\ &\rightarrow 9 \end{aligned}$$

How about an integer between 10 and 100, inclusive?

$$\begin{aligned} \textit{Integer} &\rightarrow \textit{NzdRest} \\ &\rightarrow 1 \textit{Rest} \\ &\rightarrow 1 \textit{Zd} \\ &\rightarrow 10 \end{aligned}$$

We are really cooking. Let's try one more. Is a number greater than 100 derivable?

$$\begin{aligned} Integer &\rightarrow Nz d Rest \\ &\rightarrow 1 Rest \\ &\rightarrow 1 Z d Rest \\ &\rightarrow 1 Nz d Rest \\ &\rightarrow 15 Rest \\ &\rightarrow 15 Z d \\ &\rightarrow 15 Nz d \\ &\rightarrow 154 \end{aligned}$$

Think about this grammar and see if you can see any problems with it. Are there integers that cannot be derived? Are there sentences that can be derived that are not integers?

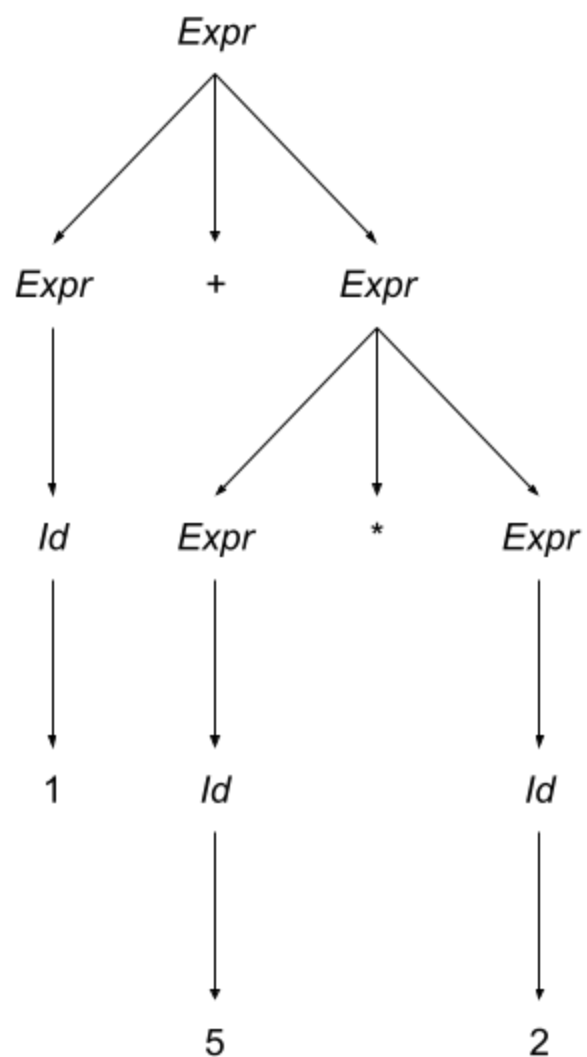
Trees or Derivations

Let's take a look at a grammar that will generate sentences that are simple mathematical expressions using the + and * operators and the numbers 0 through 9:

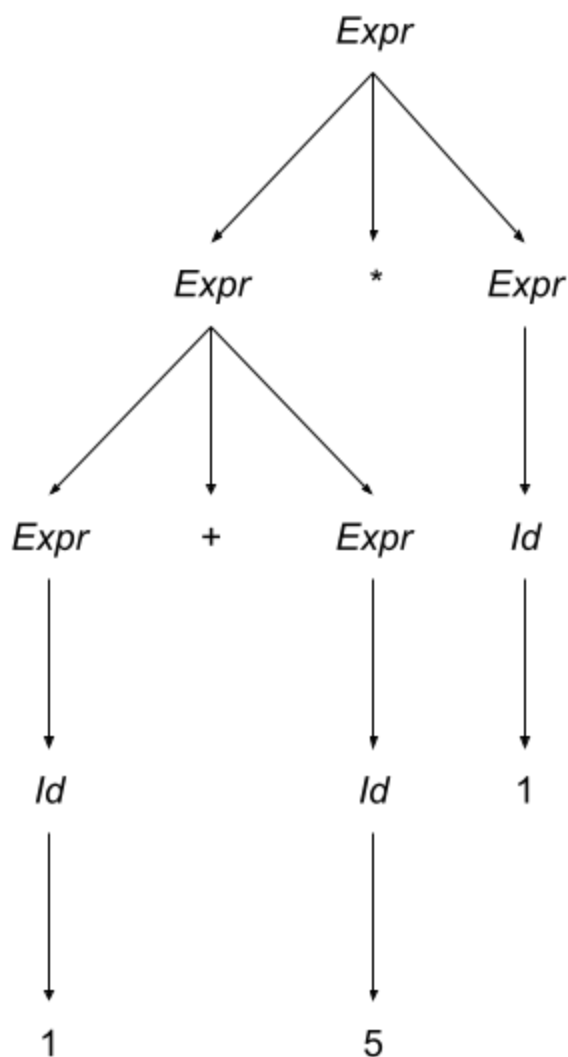
$$\begin{aligned} Expr &\rightarrow Expr + Expr \\ &| Expr * Expr \\ &| (Expr) \\ &| id \\ id &\rightarrow 0 \dots 9 \end{aligned}$$

Until now we have used proof-by-construction through derivations to show that a particular string is in the language described by a particular grammar. What's really cool is that any derivation can also be written in the form of a tree. The two representations contain the same information -- a proof that a particular sentence is in a language. Let's look at the derivation of the expression 1 +

5 * 2:



There's only one problem with this derivation: Our choice of which production to apply to expand the start symbol was arbitrary. We could have just as easily used the second production and derived the expression $1 + 5 * 2$:



This is clearly not a good thing. We do not want our choice of productions to be arbitrary. Why? When the choice of the production to expand is arbitrary, we cannot "encode" in the grammar any semantic meaning. (We will learn how to encode that semantic meaning below). A grammar that produces two or more valid parse trees for a particular sentence is known as an *ambiguous grammar*.

Let Me Be Clear

The good news is that we can rewrite the ambiguous grammar for mathematical expressions from above and get an unambiguous grammar. Like the way that we rewrote the grammar for integers, rewriting this grammar will involve adding additional productions:

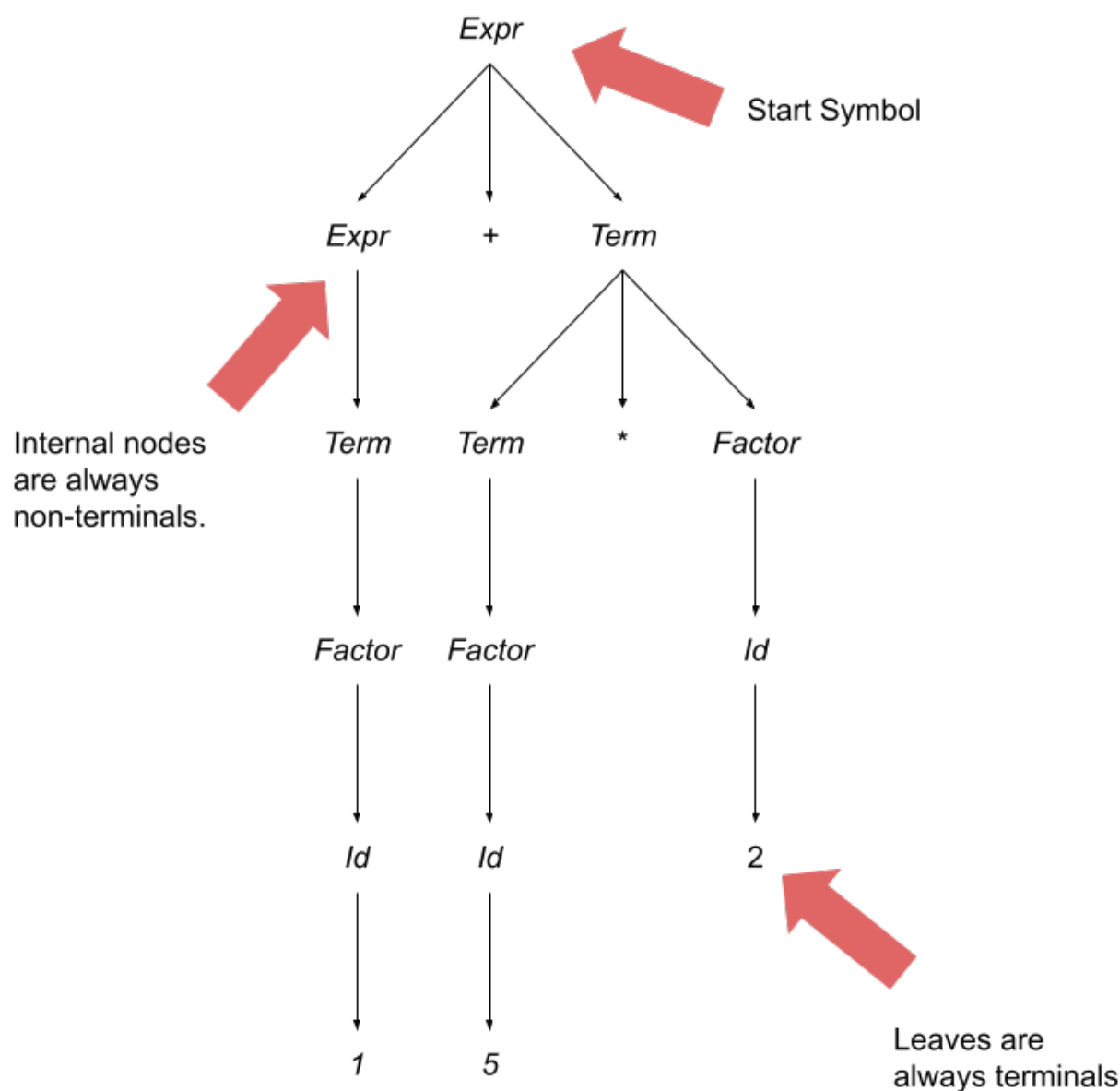
$$Expr \rightarrow Expr + Term \mid Term$$

$$Term \rightarrow Term * Factor \mid Factor$$

$$Factor \rightarrow id \mid (Expr)$$

$$id \rightarrow 0 \dots 9$$

Using this grammar we have the following parse tree for the expression $1 + 5 * 2$:



Before we think about how to encode semantic meaning in to a parse tree, let's talk about the properties of a parse tree. The root node of the parse tree is always the start symbol of the grammar. The internal nodes of a parse tree are always non-terminals. The leaves of a parse tree are always terminals.

Making Meaning

If we adopt a convention that reading a parse tree occurs through a depth-first, in-order traversal, we can add meaning to these beasts and their associated grammars. First, we can see how associativity is encoded: By writing all the productions so that "recursion" happens on the left side of any terminals (a so-called *left-recursive production*), we will arrive at a grammar that is left-

associative. The converse is true -- the productions whose recursion happens on the right side of any terminals (a *right-recursive* production) specifies right associativity. Second, we can see how precedence is encoded: The further away a production is from the start symbol, the higher the precedence. In other words, the precedence is inversely proportional to the distance from the start symbol. All alternate options for the same production have equal precedence.

Let's look back at our grammar for *Expr*. A *Term* and an addition operation have the same precedence. A *Factor* and a multiplication operation have the same precedence. The production for an addition operation is left-recursive and, therefore, the grammar specifies that addition is left associative. The same is true for the multiplication operation.

Although we said that what we are studying in this module is the *syntax* of a language and not its *semantics*, a language may have *static semantics* that the compiler can check during syntax analysis. Syntax analysis is done using a tool known as the *parser*.

Remember that syntax and syntax analysis is only concerned with valid programs. If the program, considered as a string of letters of the language's alphabet, can be derived from the language grammar's start symbol, then the program is valid. We all agreed that was the limit of what a syntax analyzer could determine.

One form of output of a parser is a parse tree. It is possible to apply "decoration" to the parse tree in order to verify certain extra-syntactic properties of a program at the time that it is being parsed. These properties are known as a language's *static semantics*. More formally, Sebesta defines static semantics as the rules for a valid program in a particular language that are difficult (or impossible!) to encode in a grammar. Checking static semantics early in the compilation process is incredibly helpful for programmers as they write their code and allows the stages of the compilation process after syntax analysis to make more sophisticated assumptions about a program's validity.

One example of static semantics of a programming language has to do with its type system. Checking a program for type safety is possible at the time of syntax analysis using an *attribute grammar*. An attribute grammar is an extension of a CFG that adds (static semantic) information to its terminals and nonterminals. This information is known as attributes. Attached to each production are attribute calculation functions. At the time the program is parsed, the attribute calculation functions are evaluated every time that the associated production is used in a derivation and the results of that invocation are stored in the nodes of the parse tree that represent terminals and nonterminals. Additionally, in an attribute grammar, each production can have a set of *predicates*. Predicates are simply Boolean-valued functions. When a parser attempts to use a production during parsing, it's not just the attribute calculation functions that are invoked -- the production's predicates are too. If any of those predicates returns false, then the derivation fails.

An Assignment Grammar

The following is a snippet of an industrial-strength grammar -- the one for the C programming language. The snippet is concerned with the assignment expression:

```
(6.5.2) postfix-expression:
    primary-expression
    postfix-expression [ expression ]
    postfix-expression ( argument-expression-list_opt )
    postfix-expression . identifier
    postfix-expression -> identifier
    postfix-expression ++
    postfix-expression --
    ( type-name ) { initializer-list }
    ( type-name ) { initializer-list , }
```

```
(6.5.1) primary-expression:
    identifier
    constant
    string-literal
    ( expression )
```

```
(6.5.3) unary-expression:
    postfix-expression
    ++ unary-expression
    -- unary-expression
    unary-operator cast-expression
    sizeof unary-expression
    sizeof ( type-name )
```

There are lots of details in the grammar for an assignment statement in C and not all of them pertain to our discussion. So, instead of using that grammar of assignment statements, we'll use a simpler one:

$$Assign \rightarrow Var = Expr$$

$$Expr \rightarrow Var + Var$$

$$Expr \rightarrow Var$$

$$Var \rightarrow A|B|C$$

Assume that these productions are part of a larger grammar for a complete, statically typed programming language. In the subset of the grammar that we are going to work with, there are three terminals: **A**, **B** and **C**. These are variable names available for the programmer in our language. As a program in this language is parsed, the compiler builds a mapping between variables and their types and keeps them in something known as the *symbol table*. An entry is added to the symbol table for each new variable declaration the parser encounters. The compiler can "lookup" the type of a variable using its name thanks to the symbol table's *lookup* function.

Our hypothetical language contains only two numerical types: **int** and **real**. A variable can only be assigned the result of an expression if that expression has the same type as the variable. In order to determine the type of an expression, our language adheres to the following rules:

- (a) If the expression is an addition of two variables, its type is
 - i. the same as the type of the variables when the variables have the same type;
 - ii. `real` otherwise.
- (b) If the expression is a single variable, its type is the type of the variable.

Let's assume that we are looking at the following program written in our hypothetical language:

```
int A;  
real B;  
A = A + B;
```

The declarations of the variables `A` and `B` are handled by parts of the grammar that we are not showing in our example. Again, when those declarations are parsed, an entry is made in the symbol table so that variable names can be mapped to types. Let's derive `A = A + B;` using the snippet of the grammar shown above:

$$\begin{aligned} Assign &\rightarrow Var = Expr \\ &\rightarrow A = Expr \\ &\rightarrow A = Var + Var \\ &\rightarrow A = A + Var \\ &\rightarrow A = A + B \end{aligned}$$

Great! The program passes the syntactic analysis so it's valid!



Right?

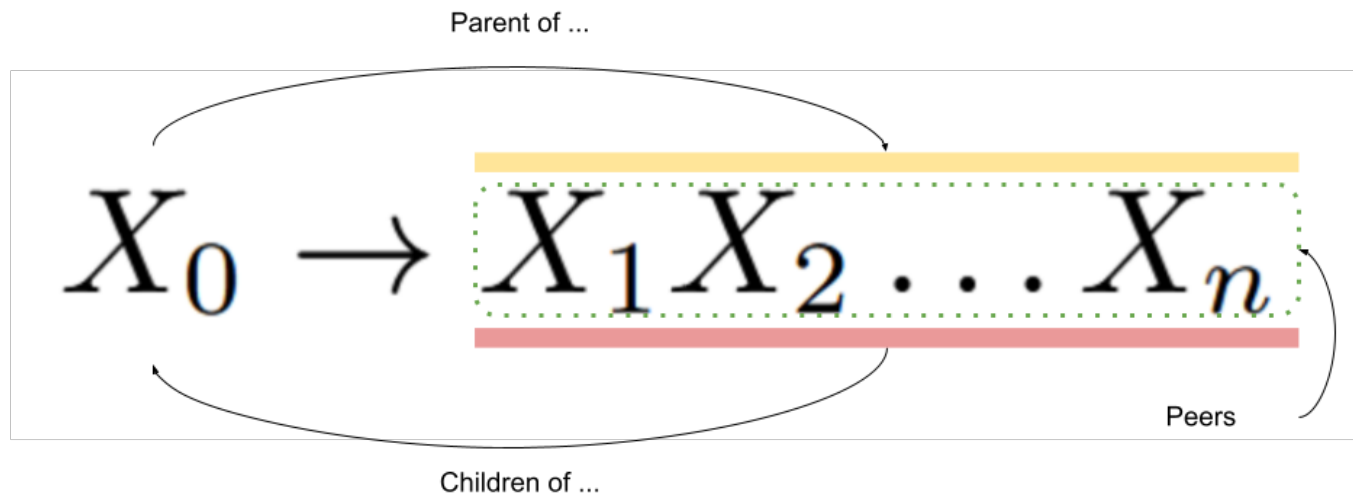
This Grammar Goes To 11

Wrong. According to the language's type rules, we can only assign to variables that have the same type as the expression being assigned. The rules say that `A + B` is a `real` (a.ii). `A` is declared as an `int`. So, even though the program parses, it is invalid!

We can detect this error using attribute grammars and that's exactly what we are going to do! For our Assign grammar, we will add the following attributes to each of the terminals and nonterminals:

- `expected_type`: The type that is expected for the expression.
- `actual_type`: The actual type of the expression.

Again, the values for the attributes are set according to *attribute calculation functions*. An attribute whose calculation function uses attribute values from only its children and peer nodes in the parse tree is known as a *synthesized attribute*. An attribute whose calculation function uses only attribute values from its parent nodes in the parse tree is known as an *inherited attribute*.



Let's define the attribute calculation functions for the `expected_type` and `actual_type` attributes of the Assign grammar:

$$\begin{aligned} \text{Assign} &\rightarrow \text{Var} = \text{Expr} \\ &\Rightarrow \text{Expr.expected_type} = \text{Var.actual_type} \end{aligned}$$

For this production, we can see that the expression's `expected_type` attribute is defined according to the variable's `actual_type` which means that, from the perspective of the expression, it is an inherited attribute.

$$\begin{aligned} \text{Expr} &\rightarrow \text{Var} \\ &\Rightarrow \text{Expr.actual_type} = \text{Var.actual_type} \end{aligned}$$

For this production, we can see that the expression's `actual_type` attribute is defined according to the variable's `actual_type` which means that, again from the expression's perspective, it is a synthesized attribute.

And now for the most complicated (but not complex) attribute calculation function definition:

$$\begin{aligned} \text{Expr} &\rightarrow \text{Var} + \text{Var} \\ &\Rightarrow \begin{cases} \text{Expr.actual_type} = \text{int} & \text{Var}_l.\text{actual_type} = \text{Var}_r.\text{actual_type} = \text{Var}_l.\text{actual_type} = \text{int} \\ \text{Expr.actual_type} = \text{real} & \text{otherwise} \end{cases} \end{aligned}$$

Again, we can see that the expression's `actual_type` attribute is defined according to its children

nodes -- the `actual_type` of the two variables being added together -- which means that it is a synthesized attribute.

If you are thinking that the attribute calculation functions are recursive, you are exactly right! And, you can probably see a problem straight ahead. So far the attribute calculation functions have relied on attributes of peer, children and parent nodes in the parse tree to already have values. Where is our base case?

Great question. There's a third type of attribute known as an *intrinsic attribute*. An intrinsic attribute is one whose value is calculated according to some information outside the parse tree. In our case, the `actual_type` attribute of a variable is calculated according to the mapping stored in the symbol table and accessible by the *lookup* function that we defined above.

$$\begin{aligned} Var &\rightarrow A|B|C \\ \Rightarrow Expr.actual_type &= lookup(A|B|C) \end{aligned}$$

That's all for the definition of the attribute calculation functions and is all well and good.

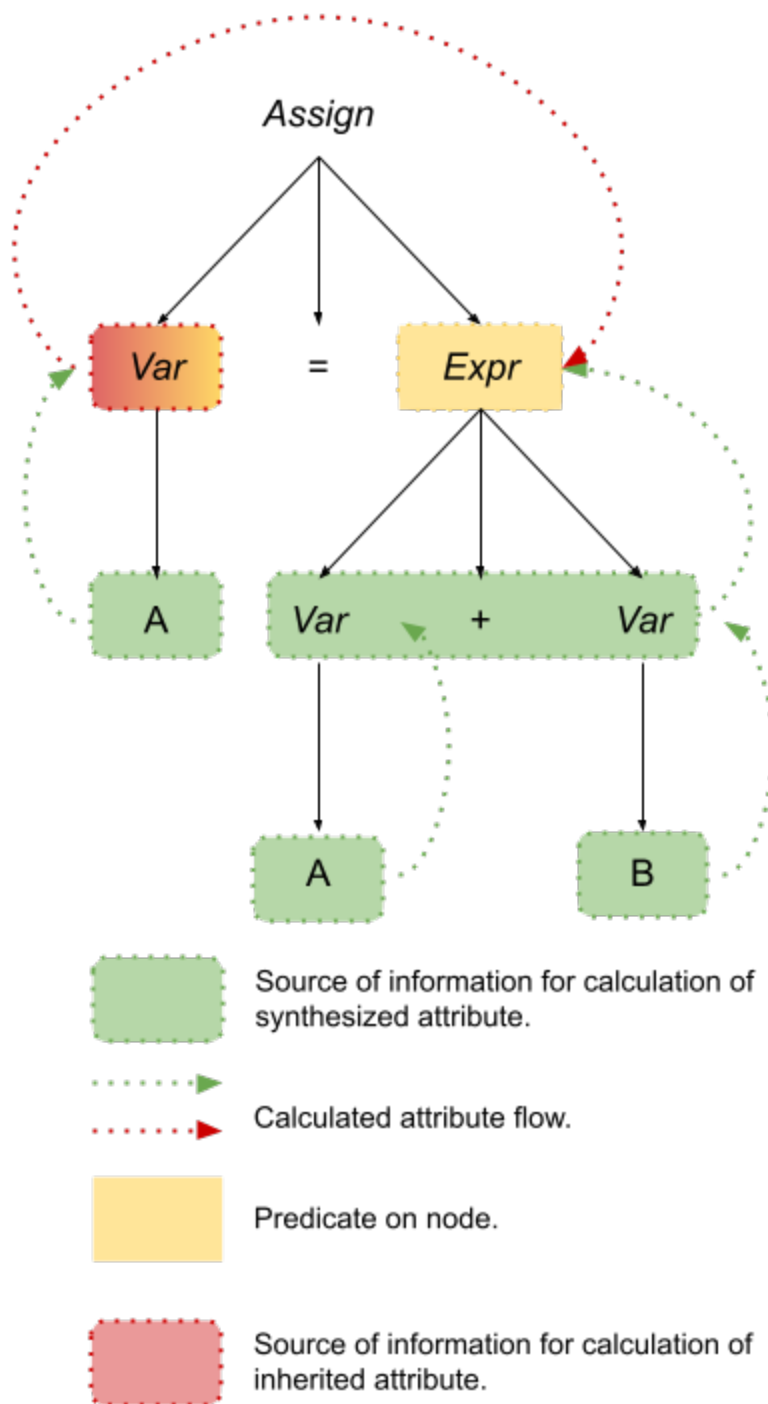
Unfortunately, we still have not used our attributes to inform the parser when a derivation has gone off the rails by violating the type rules. We'll define these guardrail predicate functions and show the complete attribute grammar at the same time:

$$\begin{aligned} Assign &\rightarrow Var = Expr \\ \Rightarrow Expr.expected_type &= Var.actual_type \\ Expr &\rightarrow Var + Var \\ \Rightarrow \begin{cases} Expr.actual_type = \text{int} & Var_l.actual_type = Var_r.actual_type = Var_l.actual_type = \text{int} \\ Expr.actual_type = \text{real} & \text{otherwise} \end{cases} \\ \checkmark Expr.actual_type &= Expr.expected_type \\ Expr &\rightarrow Var \\ \Rightarrow Expr.actual_type &= Var.actual_type \\ \checkmark Expr.actual_type &= Expr.expected_type \\ Var &\rightarrow A|B|C \\ \Rightarrow Expr.actual_type &= lookup(A|B|C) \end{aligned}$$

The equalities after the checkmarks are the predicates. We can read them as "If the actual type of the expression is the same as the expected type of the predicate, then the derivation (parse) can continue. Otherwise, it must fail because the assignment statement being parsed violates the type rules."

Put The Icing On The Cookie

The process of calculating the attributes of a production during parsing is known as *decorating* the parse tree. Let's look at the parse tree from the assignment statement `A = A + B;` and see how it is decorated:




The Daily PL - 6/15/2023

Recognizing vs Generating

From one end of the PL football stadium, grammars are generators; from the other endzone they are recognizers. Generators are tools that will, you know, *generate* things. Recognizers are tools that will, ..., recognize things. Grammars *generate* strings that are in the language that the grammar specifies. Parsers (generated from grammars) can recognize strings that are in the language that the grammar specifies.

Again, we have seen the power of the derivation and the parse tree to *generate* strings that are in a language L defined by a grammar G . We can create a valid sentence in language L by repeated application of the productions in G to the sentential forms derived from G 's start symbol. Applying all the productions in all possible combinations will eventually enumerate all valid strings in the language L (don't hold your breath for that process to finish!).

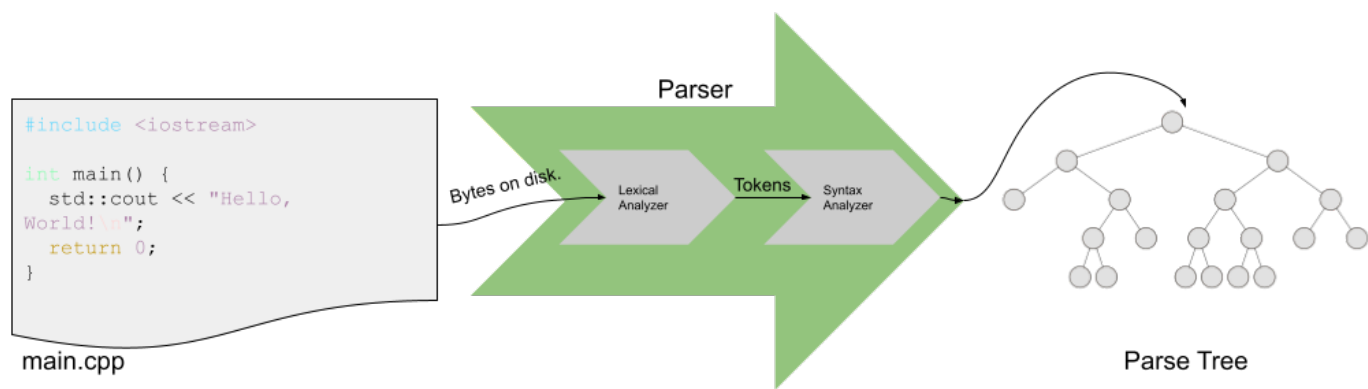
The only problem is that (**with one modern exception**  [\(https://copilot.github.com/\)](https://copilot.github.com/)), our compilers don't actually generate source code for us! It is the programmer -- us! -- who writes the source code and the compiler that checks whether it is a valid program. There are obviously myriad ways in which a programmer can write an *invalid* program in a particular programming language and the compiler can detect all of them. However, the easiest invalid programs for the compiler to detect are the ones that are not syntactically correct.

To reiterate, (most) programming languages specify their syntax using a (context-free) grammar (CFG) -- the theoretical language L that we've talked about as being defined by a grammar G can actually be a programming language! Therefore, the source code for a program is technically just a particular sequence of terminals from L 's alphabet. For that sequence of terminals to be a valid program, it must be the final sentential form in some derivation following the productions of G -- in other words, it must be a *sentence*.

We can draw a conclusion from the preceding discussion: the compiler is not a *generator* but rather a *recognizer*.

Parsers

Recall from Chapter 2 of Sebesta (or earlier programming courses), the stages of compilation. The stage/tool that recognizes whether a particular sequence of terminals from a language's alphabet is a valid program or not (a.k.a., the recognizer) is called *parsing/a parser*. Besides recognizing whether a program is valid, parsers convert the source code for a program written in a particular programming language defined according to a specific grammar into a parse tree.



What's sneaky is that the parsing process is really two processes: lexical analysis and syntax analysis. *Lexical analysis* turns the bytes on the disk into a sequence of *tokens* (and associated *lexemes*). *Syntax analysis* turns the tokens into a parse tree.

We've seen several examples of languages defined by grammars and those grammars contain productions, terminals and nonterminals. We haven't seen any tokens, though, have we? Well, tokens are an abstract way to represent groups of bytes in a file of source code in an abstract manner. The actual bytes that are in a group that are bundled together stay associated with the token and are known as *lexemes*. Tokenizing the input makes the syntax analysis process easier.

Note: Read Sebesta's discussion about the reason for separating lexical analysis from syntax analysis in Section 4.1 on (approximately pg. 143).

Turtles All The Way Down

Remember the **Chomsky Hierarchy** (https://uc.instructure.com/courses/1610326/pages/the-daily-pl-11-slash-17-slash-2021?module_item_id=66833307) of languages? Context-free languages can be described by CFGs. Slightly less complex languages are known as regular languages.

Remember from our proof of whether the language of all palindromes is regular, the defining characteristic of whether a language is regular or not is whether it can be recognized using a finite amount of memory. We can write out the abstract machine with finite memory to recognize any regular language -- that abstract machine is known as a *finite automata* (FA).

If you have ever written a regular expression then you have actually written a stylized FA. Cool, right?? You will learn more about FAs in CS4040, but for now it's important to know the broad outlines of the definition of an FA because of the central role they play in lexical analysis. An FA is

1. A (finite) set of states, S ;
2. A (finite) alphabet, A ;
3. A transition function, $f : S, A \rightarrow S$, that takes two parameters (the current state [an element in S] and an element from the alphabet) and returns a new state;
4. A start state (one of the states in S); and

5. A set of accepting states (a subset of the states in S).

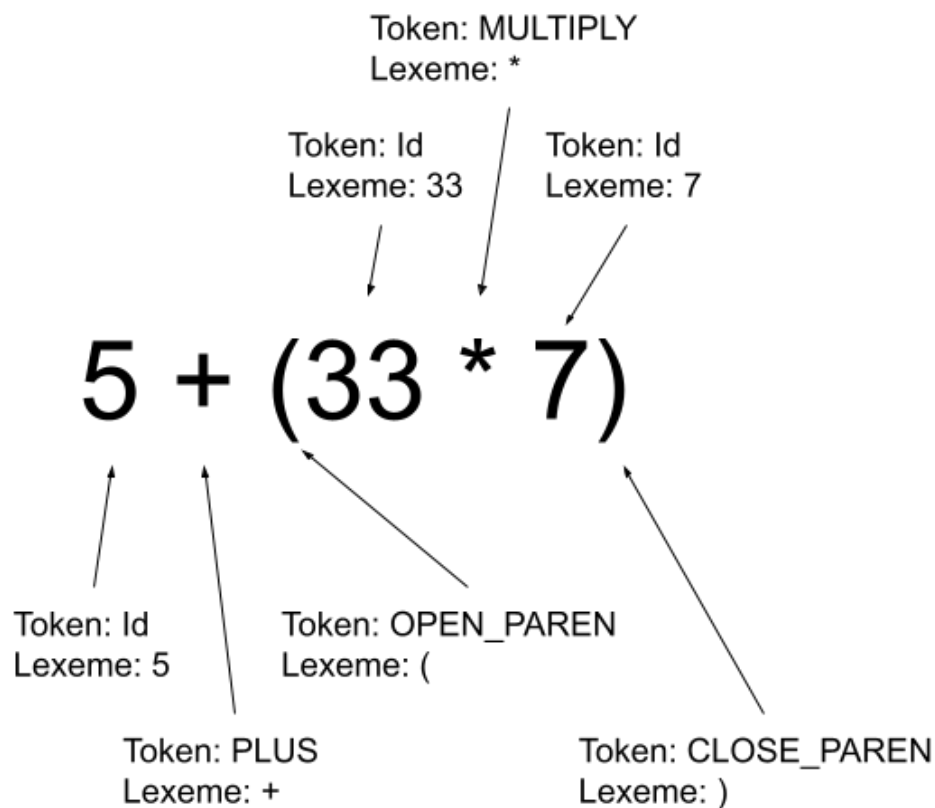
Why does this matter? Because we can describe how to group the bytes on disk into tokens (and lexemes) using regular languages. You probably have a good idea about what is going on, but let's dig in to an example -- that'll help!

Lexical Analysis of Mathematical Expressions

For the remainder of this edition, let's go back to the (unambiguous) grammar of mathematical expressions:

$$\begin{aligned}Expr &\rightarrow Expr + Term \mid Term \\Term &\rightarrow Term * Factor \mid Factor \\Factor &\rightarrow id \mid (Expr) \\id &\rightarrow 0 \dots 9\end{aligned}$$

Here's a syntactically correct expression with labeled tokens and lexemes:



What do you notice? Well, the first thing that I notice is that in most cases, the lexeme value is actually, completely, utterly *useless*. For instance, what other logical grouping of bytes other than the one that represents the `)` will be converted in to the `CLOSE_PAREN` token?

There is, however, one token whose lexeme is *very* important: the `Id` token. Why? Because that token can be any number! The actual lexeme of that `Id` makes a big difference when it comes time to actually evaluate the mathematical expression that we are parsing (if that is, indeed, the goal of the operation!).

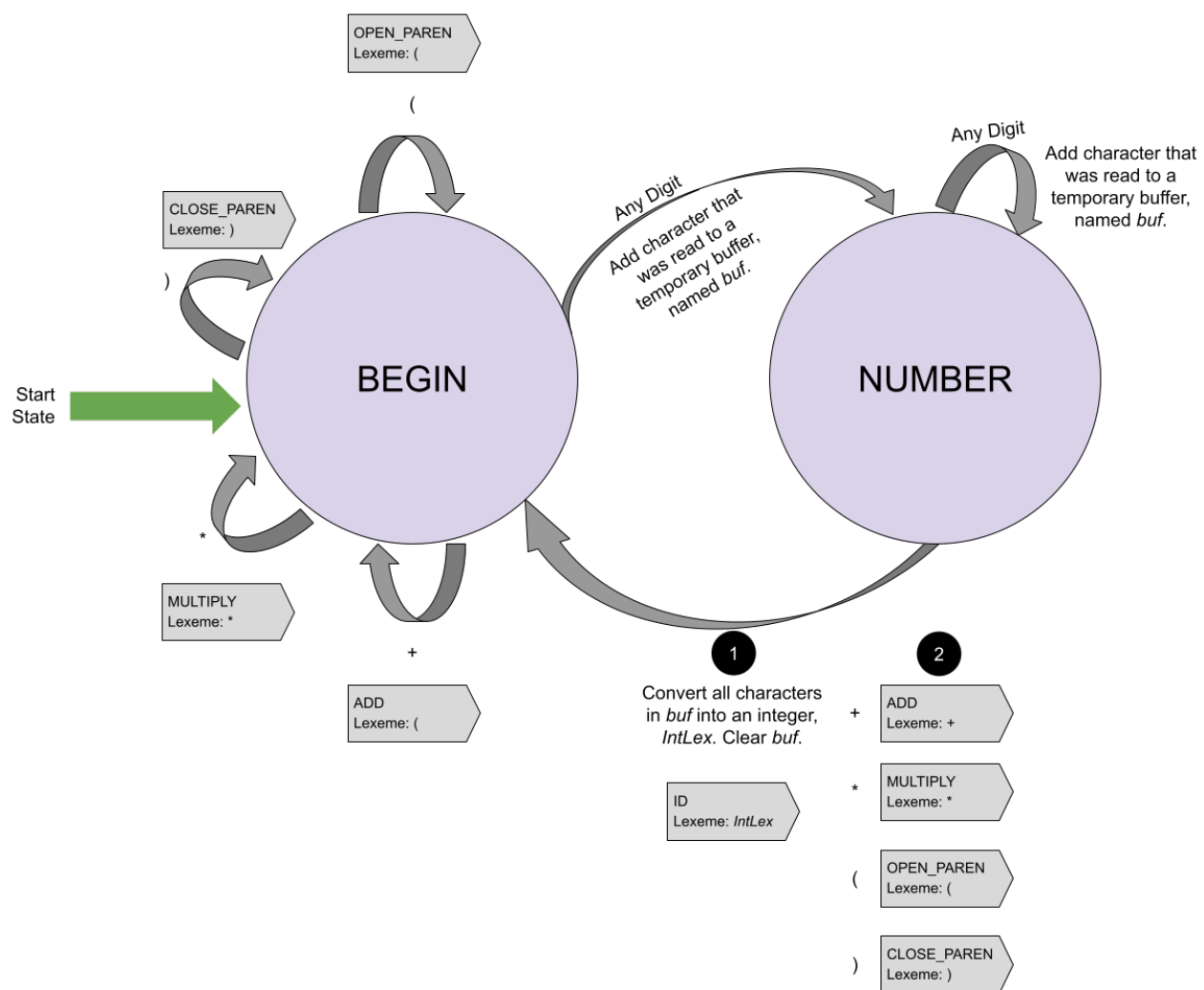
Certitude and Finitude

Let's take a stab at defining the FA that we can use to convert a stream of bytes on disk in to a stream of tokens (and associated lexemes). Let yourself slip in to a world where you are nothing more than a robot with a simple memory: you can be in one of two states. Depending on the state that you are in, you are able to perform some specific actions depending on the next character that you see. The two states are `BEGIN` and `NUMBER`. When you are in the `BEGIN` state and you see a `)`, `+`, `*`, or `(` then you simply emit the appropriate token and stay in the `BEGIN` state. If you are in the `BEGIN` state and you see a digit, you simply store that digit and then change your state to `NUMBER`. When you are in the `NUMBER` state, if you see a digit, you store that digit after all the digits you've previously seen and stay in the same state. However, when you are in the `NUMBER` state and you see a `)`, `+`, `*`, or `(`, you must perform three actions:

1. Convert the string of digits that you are remembering into a number and emit a `Id` token with the converted value as the lexeme;
2. Emit the token appropriate to the value you just saw; and
3. Reset your state to `BEGIN`.

For as long as there are bytes in the input file, you continue to perform those operations robotically. The operations that I just described are the FA's state transition function! See how each operation depends only on the current state and an input? Those are the parameters to the state transition function! And see how we specified the state we will be in after considering the input? That's the function's output! Woah!

What's amazing about all those words that I just wrote is that they can be turned into a picture that is far easier to understand:




Make the connection between the different aspects of the written and graphical description above and the formal components of an FA, especially the state transition function. In the image above, the state transition function is depicted with the gray arrows!

The Last Step...

There's just one more step ... we want to convert this high-level description of a tokenizer into actual code. And, we can! What's more, we can use a technique we learned earlier this semester in order to do it! Consider that the tokenizer will work hand-in-hand with the syntax analyzer. As the syntax analyzer goes about its business of creating a parse tree, it will occasionally turn to the lexical analyzer and say, "Give me the next token!". In between answers, it would be helpful if the tokenizer could maintain its state.

If that doesn't sound like a coroutine, then I don't know what does! Because a coroutine is *dormant* and *not dead* between invocations, we can easily program the tokenizer as a coroutine so that it remembers its state (either `BEGIN` or `NUMBER`) and other information (like the current digits that it has seen [if it is in the `NUMBER` state] and the progress it has made reading the input file). Kismet!

To see such an implementation in Python, check [here](https://github.com/hawkinsw/cs3003/blob/main/recursive_descent/lexer/tokenizer.py)  (https://github.com/hawkinsw/cs3003/blob/main/recursive_descent/lexer/tokenizer.py).

Peanut Butter and Jelly

To reiterate, the goal of a parser is to convert the source code for a program written in a particular programming language defined according to a specific grammar into a parse tree. Parsing occurs in two parts: lexical analysis and syntax analysis. The lexical analyzer converts the program's source code (in the form of bytes on disk) into a sequence of tokens. The syntax analyzer, turns the sequence of tokens in to a parse tree. The lexical analyzer and the syntax analyzer go hand-in-hand: As the syntax analyzer goes about its business of creating a parse tree, it will periodically turn to the lexical analyzer and say, "Give me the next token!".

We saw algorithms for building a lexical analyzer directly from the language's CFG. It would be great if we had something similar for the syntax analyzer. In today's lecture we are going to explore just one of the many techniques for converting a CFG into code that will build an actual parse tree. There are many such techniques, some more general and versatile than others. Sebesta talks about several of these. If you take a course in compiler construction you will see those general techniques defined in detail. In this class we only have time to cover one particular technique for translating a CFG into code that constructs parse trees and it only works for a subset of all grammars.

With those caveats in mind, let's dig in to the details!

Descent Into Madness

A *recursive-descent parser* is a type of parser that can be written directly from the structure of a CFG -- as long as that CFG meets certain constraints. In a recursive descent parser built from a CFG G , there is a subprogram for every nonterminal in G . Most of these subprograms are

recursive. A recursive-descent parser builds a parse tree from the top down, meaning that it begins with G 's start symbol and attempts to build a parse tree that represents a valid derivation whose final sentential form matches the program's tokens. There are other parsers that work bottom up, meaning that they start by analyzing a sentential form made up entirely of the program's tokens and attempt to build a parse tree that that "reduces" to the grammar's start symbol. That the subprograms are *recursive* and that the parse tree is built *top down*, the recursive-descent parser is aptly named.

We mentioned before that there are limitations on the types of languages that a recursive-descent parser can recognize. In particular, recursive-descent parsers can only recognize *LL* grammars. Those are grammars whose parse trees represent a leftmost derivation and can be built from a single left-to-right scan of the input tokens. To be precise, the first L represents the left-to-right scan of the input and the second L indicates that the parser generates a leftmost derivation. There is usually another restriction -- how many lookahead tokens are available. A lookahead token is the *next* token that the lexical analyzer will return as it scans through the input. Limiting the number of lookahead tokens reduces the memory requirements of the syntax analyzer but restricts the types of languages that those syntax analyzers can recognize. The number of lookahead tokens is written after LL and in parenthesis. *LL(1)* indicates 1 token of lookahead. We will see the practical impact of this restriction later in this edition.

All of these words probably seem very arbitrary, but I think that an example will make things clear!

Old Faithful

Let's return to the grammar for mathematical expressions that we have been examining throughout this module:

$$\begin{aligned}Expr &\rightarrow Expr + Term \mid Term \\Term &\rightarrow Term * Factor \mid Factor \\Factor &\rightarrow id \mid (Expr) \\id &\rightarrow 0 \dots 9\end{aligned}$$

We will assume that there are appropriately named tokens for each of the terminals (e.g, the token is `CLOSE_PAREN`) and that any numbers are tokenized as `ID` with the lexeme set appropriately.

According to the definition of a recursive-descent parser, we want to write a (possibly recursive)

subprogram for each of the nonterminals in the grammar. The job of each of these subprograms is to parse the the upcoming tokens into a parse tree that matches the nonterminal. For example, the (possibly recursive) subprogram for *Expr*, `Expr`, parses the upcoming tokens into a parse tree for an expression and returns that parse tree. To facilitate recursive calls among these subprograms, each subprogram returns the root of the parse tree that it builds. The parser begins by invoking the subprogram for the grammar's start symbol. The return value of that function call will be the root node of the parse tree for the entire input expression. Any recursive calls to other subprograms from within the subprogram for the start symbol will return *parts* of that overall parse tree.

I am sure that you see why each of the subprograms usually contains some recursive function calls -- the nonterminals themselves are defined recursively.

How would we write such a (possibly recursive) subprogram to build a parse tree rooted at a *Factor* from a sequence of tokens?

There are two productions for a *Factor* so the first thing that the `Factor` subprogram does is determine whether it is parsing, for example, $(5+2)$ -- a parenthesized expression -- or `918` -- a simple `ID`. In order to differentiate, the function simply consults the lookahead token. If that token is an open parenthesis then it knows that it is going to be evaluating the production $Factor \rightarrow (Expr)$. On the other hand, if that token is an `ID`, then it knows that it is going to be evaluating the production $Factor \rightarrow id$. Finally, if the current token is neither an open parenthesis nor an `ID`, then that's an error!

Let's assume that the current token is an open parenthesis. Therefore, `Factor` knows that it should be parsing the production $Factor \rightarrow (Expr)$. Remember how we said that in a recursive-descent parser, each nonterminal is represented by a (possibly recursive) subprogram? Well, that means that we can assume there is a subprogram for parsing an *Expr* (though we haven't yet defined it!). Let's call that mythical subprogram `Expr`. As a result, the `Factor` subprogram can invoke `Expr` which will return a parse tree rooted at that expression. Pretty cool! Before continuing to parse, `Factor` will check `Expr`'s return value -- if it is an error node, then parsing stops and `Factor` simply returns that error.

Otherwise, after successfully parsing an expression (by invoking `Expr`) the parser expects the next token to be a close parenthesis. So, `Factor` checks that fact. If everything looks good, then `Factor` simply returns the node generated by `Expr` -- there's no need to generate another node that just indicates an expression is wrapped in parenthesis. If the token after parsing the expression is not a close parenthesis, then `Factor` returns an error node.

Now, what happens if the lookahead token is an `ID`? That's simple -- `Factor` will just generate a node for that `ID` and return it!

Finally, if neither of those is true, `Factor` simply returns an error node.

Let's make this a little more concrete by writing that in pseudocode. We will assume the following are defined:

1. `Node(T, X, Y, Z ...)`: A polymorphic function that generates an appropriately typed node (according to `T`) in the parse tree that "wraps" the tokens `X`, `Y`, `Z`, etc. We will play fast and loose with this notation.
2. `Error(X)`: A function that generates an error node because token `X` was unexpected -- an error node in the final parse tree will generate a diagnostic message.
3. `tokenizer()`: A function that returns and consumes the next token from the lexical analyzer.
4. `lookahead()`: A function that returns the lookahead token.

```
def Factor:
    if lookahead() == OPEN_PAREN:
        # Parsing a ( Expr )
        #
        # Eat the lookahead and move forward.
        curTok = nextToken()
        # Build a parse tree rooted at an expression,
        # if possible.
        nestedExpr = Expr()
        # There was an error parsing that expression;
        # we will return an error!
        if type(nestedExpr) == Error:
            return nestedExpr
        # Expression parsing went well. We expect a )
        # now.
        if lookahead() == CLOSE_PAREN:
            # Eat that close parenthesis.
            nextToken()
            # Return the root of the parse tree of the
            # nested expression.
            return nestedExpr
        else:
            # We expected a ) and did not get it.
            return Error(lookahead())
    else if lookahead() == ID:
        # Parsing a ID
        #
        curTok = nextToken()
        return Node(curTok)
    else:
        # Parsing error!
        return Error(lookahead())
```

Writing a function to parse a ***Factor*** is relatively straightforward. To get a sense for what it would be like to parse an expression, let's write a part of the `Expr` subprogram:

```
def Expr:
    ...
    leftHandExpr = Expr()
    if type(leftHandExpr) == Error:
        return leftHandExpr
    if lookahead() != PLUS:
        curTok = nextToken()
        return Error(curTok)
    rightHandTerm = Term()
    if type(rightHandTerm) == Error:
```

```

    return rightHandTerm
  return Node(Expr, leftHandExpr, rightHandTerm)
  ...

```

What stands out is an emerging pattern that each of the subprograms will follow. Each subprogram is slowly matching the items from the grammar with the actual tokens that it sees. The subprogram associated with each nonterminal parses the nonterminals used in the production, "eats" the terminals in those same productions and converts it all into nodes in a parse tree. The subprogram that calls subprograms recursively melds together their return values into a new node that will become part of the overall parse tree, one level up.

We Are Homefree

I don't know about you, but I think that's pretty cool -- you can build a series of collaborating subprograms named after the nonterminals in a grammar that call each other and, bang!, through the power of recursion, a parse tree is built! I think we're done here.

Or are we?

Look carefully at the definition of `Expr` given in the pseudocode above. What is the name of the first subprogram that is invoked? That's right, `Expr`. When we invoke `Expr` again, what is the first subprogram that is invoked? That's right, `Expr` again. There's no base case -- this spiral will continue until there is no more space available on the stack!

It seems like we may have run head-on into a fundamental limitation of recursive-descent parsing: The grammars that it parses cannot contain productions that are *left recursive*. A production $A \rightarrow \dots$ is (indirect) left recursive "when A derives itself as its leftmost symbol using one or more derivations." In other words, $A \rightarrow^+ A \dots$ is indirectly left recursive where \rightarrow^+ indicates *one or more* productions. For example, the production for A in grammar

$$\begin{aligned} A &\rightarrow B|a \\ B &\rightarrow Ab|b \end{aligned}$$

is *indirect left recursive* because $A \rightarrow B \rightarrow Ab$.

A production $A \rightarrow \dots$ is direct left recursive when A derives itself as its leftmost symbol using *one* derivation (e.g., $A \rightarrow A \dots$). The production for *Expr* in our working example is direct left recursive.

What are we to do?

Note: The definitions of left recursion are taken from:

Allen B Tucker and Robert Noonan. 2006. *Programming Languages* (2nd. ed.). McGraw-Hill, Inc.,

USA.

Formalism To The Rescue

Stand back, we are about to deploy math!

π
STAND
BACK
 I'M
TRYING
MATH

There is an algorithm for converting productions that are direct-left recursive into productions that generate the same languages and are not direct-left recursive. In other words, there is hope for recursive-descent parsers yet! The procedure is slightly wordy, but after you see an example it will make perfect sense. Here's the overall process:

1. For the rule that is direct-left recursive, A , rewrite all productions of A as $A \rightarrow A\alpha_1 | A\alpha_2 | \dots | A\alpha_n | \beta_1 | \dots | \beta_n$ where all (non)terminals $\beta_1 \dots \beta_n$ are not direct-left recursive.
2. Rewrite the production as

$$\begin{aligned}
 A &\rightarrow \beta_1 A' | \beta_2 A' | \dots | \beta_n A' \\
 A' &\rightarrow \alpha_1 A' | \alpha_2 A' | \dots | \alpha_n A' | \varepsilon
 \end{aligned}$$

where ε is the *erasure rule* and matches an empty token.

I know, that's hard to parse (pun intended). An example will definitely make things easier to grok:

In

$$Expr \rightarrow Expr + Term | Term$$

A is $Expr$, α_1 is $+Term$, β_1 is $Term$. Therefore,

$$\begin{aligned}
 A &\rightarrow \beta_1 A' \\
 A' &\rightarrow \alpha_1 A' | \varepsilon
 \end{aligned}$$

becomes


$$\begin{aligned}
 Expr &\rightarrow Term Expr' \\
 Expr' &\rightarrow +Term Expr' | \varepsilon
 \end{aligned}$$

No sweat! It's just moving pieces around on a chessboard!

The Final Countdown

We can make those same manipulations for each of the productions in the grammar in our working example and we arrive here:

$$\begin{aligned}Expr &\rightarrow TermExpr' \\Expr' &\rightarrow +TermExpr' | \epsilon \\Term &\rightarrow FactorTerm' \\Term' &\rightarrow *FactorTerm' | \epsilon \\Factor &\rightarrow (Expr) | id\end{aligned}$$

Now that we have a non direct-left recursive grammar we can easily write a recursive-descent parser for the entire grammar. The source code is available [online](https://github.com/hawkinsw/cs3003/tree/main/recursive_descent)  (https://github.com/hawkinsw/cs3003/tree/main/recursive_descent) and I encourage you to download and play with it!