A Practical Tutorial on Context Free Grammars

Dr. Robert B. Heckendorn
University of Idaho

March 17, 2021

Contents

A grammar is used to specify the syntax of a language. It answers the question: What sentences are in the language and what are not? **A sentence** is a finite sequence of symbols from the alphabet of the language. The grammar we discuss here is for a **context free languages**. This is the grammar used for most modern computer languages. The grammar is a **context free grammar** or **CFG** and hence can be processed by a **push down automata**. A **syntax error** for language L indicates that the stream of symbols is not producible from the grammar for L. A **scanner error** occurs when the next character input does not match any allowed token in the language. For example the character '@' might not be allowed in the language at all outside of a string. A **semantic error** happens when the the statements in the language match the grammar but the statements are not legal. An example is a mismatch type like trying to add two Booleans together or perhaps a break statement not in a loop. Side note: scanners are definable by regular expressions and hence only require a state machine to implement.

The goal of this document is to help students understand grammars as they are used in Computer Science through the use of many examples and to provide a reference of common things they might want to do with a grammar.

1 Some Definitions

A **grammar** defines what are legal statements in a language.

A grammar has:

- **Alphabet** is a finite set of symbols that appear in the language
- **Nonterminals** symbols that can be replaced by collections of symbols found in a production (see below)
- **Terminal** symbols from the alphabet
- **Productions** replacement rules for nonterminals
- **Start symbol** the initial symbol from which all sentences in the language can be derived. Note: it is usually the left hand side of the first production when a start symbol is not specifically given.

Backus-Naur Form (BNF) is a notation for expressing a CFG. The notation:

- **Nonterminals** are denoted by surrounding symbol with `<>`. e.g. `<turtle>`
- **Alternation** is denoted by `|` e.g. `bad <cats> | good <dogs>`. Strictly speaking before we introduced BNF, if you wanted:

  ```
  <animal>  ::= bad <cats> | good <dogs>
  ```

  you could say the same thing without alternation:

  ```
  <animal>  ::= bad <cats>
  <animal>  ::= good <dogs>
  ```

- **Replacement** is denoted by `::=`. These are the **productions**. The **left hand side (lhs)** of a the production is the nonterminal symbol to the left of the `::=`. The **right hand side (rhs)** of a the production is the sequence of terminal and nonterminal symbols to the right of the `::=`.

```
e.g.  <dogs> ::= corgi | other
```

- **Blanks** are ignored or must be in some escaping scheme like quotes " "
- **Terminals** are unadorned symbols

If and only if a sentence is composed of an ordered list of elements from the alphabet and it can be derived from the start symbol by application of production rules then it is in the language defined by the grammar. Specifically a **context free grammar (CFG)** is defined by a set of productions in which the left hand side of the production is a single nonterminal which may be replace by the right hand side anywhere where the left hand side occurs, regardless of the context in which the left hand side symbol occurs. Hence "context free".

Here is an example of a context free grammar using BNF for simple English sentences:

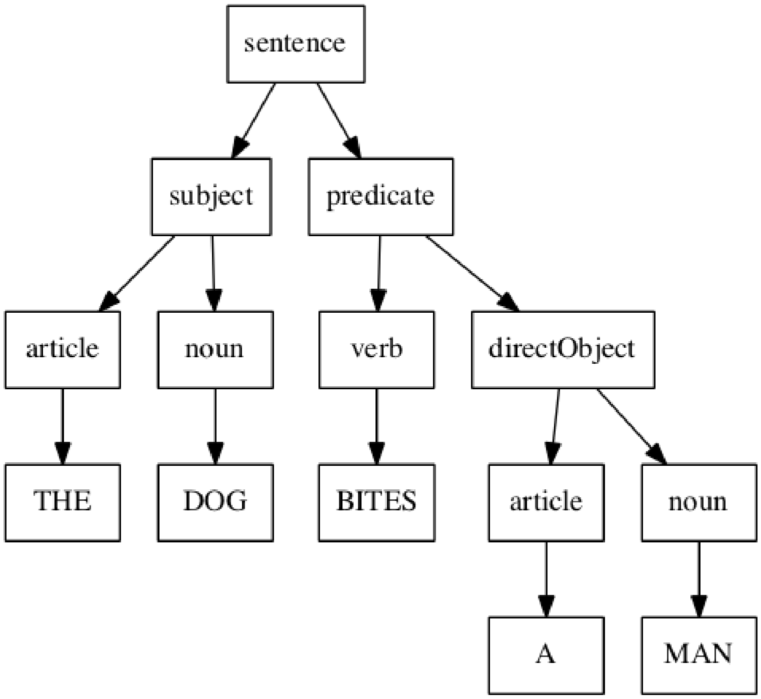
```
<sentence>       ::= <subject> <predicate>
<subject>        ::= <article> <noun>
<predicate>      ::= <verb> <direct-object>
<direct-object>  ::= <article> <noun>
<article>        ::= THE | A
<noun>           ::= MAN | DOG
<verb>           ::= BITES | PETS
```

Is the sentence "THE DOG BITES A MAN" in the language? That is does

```
<sentence> -> THE DOG BITES A MAN?
```

MAN BITES DOG is not in the grammar.

Parse tree is a tree based notation for describing the way a sentence could be built from a grammar. The root is the start symbol. the nodes are terminals and nonterminals. If a node is a nonterminal on the lhs of a production then the children of the the node are the derived terminals and nonterminals on the rhs of that production. There might be more than one parse tree for a given sentence in a language (see ambiguity). For the sentence THE DOG BITES A MAN we have the parse tree:



Derivation is the ordered list of steps used in construction of a specific parse tree for a sentence from a grammar.

Left Most Derivation is a derivation in which the left most nonterminal is always replaced first.

Parse is to show how a sentence could be built from a grammar.

Metasymbols are symbols outside the language to prevent circularity in talking about the grammar.

2 Some Common Idioms and Hierarchical Development

Many of the linguistic structures in a computer language come from a small set of basic idioms. Here are some basic forms for some common things you might want in your language. Assume start symbol: `<sentence>`.

2.1 A grammar for a language that allows a list of X's

```
<sentence>  ::= X | X <sentence>
```

This is also a grammar for that language:

```
<sentence>  ::= X | <sentence> X
```

This is an example that there can be multiple correct grammars for the same language.

2.2 A Language that is a List of X's or Y's

```
<sentence>  ::= X | Y | X <sentence> | Y <sentence>
```

Here is a more hierarchical way to do the same thing:

```
<sentence>  ::= <sub> | <sub> <sentence>
<sub>       ::= X | Y
```

Note that the first part handles this "listing" of the subparts which can either be an X or Y. It is often clarifying to do grammars hierarchically.

Here are some grammars for two similar, but different languages:

```
<sentence>  ::= X | Y | X <sentence>
```

is a grammar for one or more X's followed by an X or Y.

```
<sentence>  ::= X | Y | <sentence> X
```

is a grammar for an X or a Y followed by one or more X's. If there was no Y we could not tell which way the string was built from the final string and so the languages would be identical i.e. a list of 1 or more Xs.

2.3 A Language that is a List of X's Terminated with a Semicolon

```
<sentence>  ::= <listx> ";"
<listx>     ::= X | X <listx>
```

Note the hierarchical approach.

2.4 Hierarchical Complexification

Let's now combine the previous and make a list of sublists of Xs that end in semicolons. Note that the we describe a list of `<list>` and then describe how the sublists end in semicolons and then how to make a list. Very top to bottom and hierarchical. This will help you develop clean grammars and look for bugs such as unwanted recursion.

```
<sentence>  ::= <list> | <sentence> <list>
<list>      ::= <listx> ";"
<listx>     ::= X | X <listx>
```

2.5 A Language Where There are not Terminal Semicolons but Separating Commas

```
<listx>  ::= X | X "," <listx>
```

2.6 A language where each X is followed by a terminating semicolon:

Sometimes you need to ask yourself is this a **separating delimiter** or a **terminating delimiter**?

```
<listx>  ::= X ";" | X ";" <listx>
```

Compare again with the separating case in which each X is separated from the next:

```
<listx>  ::= X | X "," <listx>
```

This is a terminating delimiter case. Hierarchically it looks like:

```
<sentence>  ::= <sub> | <sentence> <sub>
<sub>       ::= X ","
```

2.7 An argument list of X's, Y's and Z's:

```
<arglist>  ::= "(" ")" | "(" <varlist> ")"
<varlist>  ::= <varlist> "," <var> | <var>
<var>      ::= X | Y | Z
```

2.8 The use of empty rhs ϵ

Another way to create the empty argument list is to use the ϵ or empty rhs.

```
<arglist>  ::= "(" <varlist> ")"
<varlist>  ::= <vars> | ϵ
<vars>     ::= <vars> "," <var> | <var>
<var>      ::= X | Y | Z
```

2.9 A Simple Type Declaration:

This is the grammar for a very simple C-like type declaration statement. It has a very hierarchical feel:

```
<tdecl>    ::= <type> <varlist> ";"
<varlist>  ::= <var> | <varlist> "," <var>
<var>      ::= X | Y | Z
<type>     ::= int | bool | string
```

2.10 Augment the Type Grammar with the Keyword Static

```
<tdecl>      ::= <type> <varlist> ";"
<varlist>    ::= <var> | <varlist> "," <var>
<var>        ::= X | Y | Z
<type>       ::= static <basictype> | <basictype>
<basictype>  ::= int | bool | string
```

Notice how I slipped the static as an option before the type allowing either **static** followed by the basic type or just the basic type. Hierarchy, hierarchy, hierarchy.

2.11 A Tree Structure as Nested Parentheses

For instance consider this popular nested way to represent a tree: `(ant)` or `(cat, bat)` or `((cat), (bat))` or `((cat, bat, dog), ant)`. Note the hierarchical handling of each feature. Note also that when `<tree>` points up from `<thing>` to the top of the grammar creating the nesting effect. It is important to see that matching `<tree>` requires that parentheses be removed for each time through the loop to the top. This prevents infinite recursion because you will run out of parentheses if you remove a pair each time through. You can think of it as leaving evidence of the recursion in the form of the parentheses. Note how you can immediately tell that every sentence in the languages begins and ends with a parenthesis.

```
<tree>   ::= "(" <list> ")"               // deal with parentheses
<list>   ::= <thing> | <list> "," <thing>  // do the list of things
<thing>  ::= <tree> | <name>              // things are more trees or names
<name>   ::= ant | bat | cow | dog
```

3 Associativity and Precedence

Getting the parse tree to represent proper grouping when grouping delimiters like parentheses are missing requires that we understand associativity and precedence. Modern C++ has a precedence hierarchy that is over dozen levels deep. Here is where hierarchical design again shines prominently.

3.1 A Grammar for an Arithmetic Expression

This involves the five operators $+$, $-$, $*$, $/$, $\hat{}$ (where $\hat{}$ is exponentiation). **Operator Associativity** determines the order of execution of homogeneous operators. The first four are **evaluated left to right**. That is their **associativity** is left to right or **left associative**. Exponentiation in mathematics is done right to left, that is, it is **right associative**. You can see in the grammar below by where the recursion occurs in the statement. For `<exp>` the recursion of `<exp>` occurs to the left of the operator, while for `<mulpart>` the recursion is on the right.

Operator precedence is rule for determining the order of execution of heterogeneous operators in an expression. **precedence** is handled by grouping operators of same precedence in the same production. You can see that + and - have the same precedence as does * and /. The operators with **highest precedence** occur farther down in the grammar, that is, an expression is a sum of products which is product of exponentiation.

3.2 Grouping with Parentheses

Finally, products of sums can be denoted by putting the sum in parentheses as in:

```
666*(42+496)*(x+y+z)
```

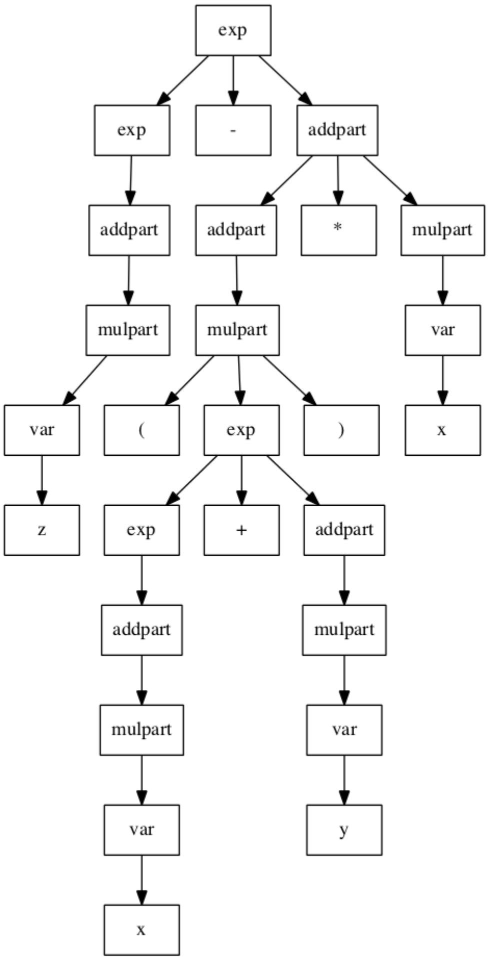
To get this affect the parenthesized expression is put in the same place that any variable could be put. In order to return to the top of the grammar (see Tips section below) the parentheses act as a counter for the number of times you return to the "top" of the grammar.

```
<exp>      ::= <exp> + <addpart> | <exp> - <addpart> | <addpart>
<addpart>  ::= <addpart> * <mulpart> | <addpart> / <mulpart> | <mulpart>
<mulpart>  ::= <group> ^  <mulpart> | <group>
<group>    ::= <var> | ( <exp> )
<var>      ::= X | Y | Z
```

Let's look at a parsing of the expression Z - (X + Y)*X with this simplified version of the above grammar:

```
<exp>      ::= <exp> + <addpart> | <exp> - <addpart> | <addpart>
<addpart>  ::= <addpart> * <mulpart> | <mulpart>
<mulpart>  ::= <var> | ( <exp> )
<var>      ::= x | y | z
```

Here is the parse tree for the statement $z - (x + y) * x$:



Since this is a context free grammar the above grammar can be simplified by carefully replacing all occurrences in the grammar of `<var>` with the right hand side of the production: `<var> ::= X | Y | Z`. This gives us the simplified grammar:

```
<exp>      ::= <exp> + <addpart> | <exp> - <addpart> | <addpart>
<addpart>  ::= <addpart> * <mulpart> | <mulpart>
<mulpart>  ::= X | Y | Z | ( <exp> )
```

3.3 Unary operators

Unary operators generally bind very tightly so they go close to the variables and grouping operators. Here we have added unary minus:

```
<exp>      ::= <exp> + <addpart> | <exp> - <addpart> | <addpart>
<addpart>  ::= <addpart> * <mulpart> | <addpart> / <mulpart> | <mulpart>
<mulpart>  ::= <unary> ^  <mulpart> | <unary>
<unary>    ::= - <unary> | <group>
<group>    ::= <var> | ( <exp> )
<var>      ::= X | Y | Z
```

Notice that we allow things like $-X$, $---X$, and $-(X)$. If we had done: `<unary> ::= - <group> | <group>` instead we would not be allowed to recursively apply a unary operator. This would prevent $--X$ from being in the language! In some circumstances this might be what you want. Implement carefully.

3.4 An Example Problem

Let's put it all together by solving a problem taken from an exam for the Computer Science Compilers Course:

> Write an unambiguous grammar for the Zev language. In the Zev language a variable is one of the letters Z, E, or V. The lowest precedence binary operator is + and it is evaluated left to right. Then there are two binary operators # and % which have equal precedence but higher precedence than +. They are also evaluated left to right and so are **left associative**. This is also the binary @ operator which is of higher precedence and is **right associative**. The * operator is a unary operator and applied on the left. Either brackets or parens can be used for grouping. For example: [Z@[E#E]], [[Z]], *[[Z]%[E]%[V]], V#**V#*V, and E are in the language.

The answer grammar is follows. Note how the nonterminals almost always point to nonterminals that appear later down the page. When they do point to nonterminals defined earlier you have to "remove" something. In out case either a pair of parentheses or brackets. This strategy means you must eventually arrive at terminals and not go on forever. Either you keep moving "down the page" or if you go back up you had to remove some nesting parens and you will run out of them.

```
<zev>    ::= <zev> + <pound> | <pound>                 // <zev> appears on left of +
<pound>  ::= <pound> # <at> | <pound> % <at> | <at>  // <pound> appears on left of #
<at>     ::= <unary>  <at> | <unary>                   // <at> appears on right of
<unary>  ::= * <unary> | <var>                         // * is applied to <unary> not <var>
<var>    ::= Z | E | V | ( <zev> ) | [ <zev> ]         // parens in same production with vars
```

4 How can Things go Wrong

The most common problems with grammars are:

- Using symbols in productions which are not in your alphabet. Don't do this.
- Inescapable productions. We will talk about this briefly.
- Ambiguous grammars. This we will discuss in great detail.

4.1 Inescapable Productions

Consider these two simple grammars:

```
<exp>  ::= <exp> + <mul> | <exp> - <mul> | <mul>
<mul>  ::= <mul> * <var> | <mul> / <var> | <var>
<var>  ::= X | Y | X
```

and

```
<exp>  ::= <exp> + <mul> | <exp> - <mul>
<mul>  ::= <mul> * <var> | <mul> / <var>
<var>  ::= X | Y | X
```

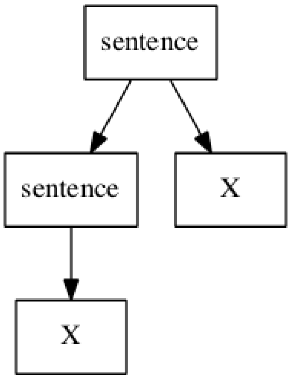
The second grammar might look reasonable, but wait... given a `<exp>` how would you ever get rid of `<exp>` since every right hand side contains an `<exp>`?! This means that you could never generate a sentence that was devoid of the nonterminal `<exp>`! The second grammar is invalid because all of the right hand side options contain the left hand side and so the production is inescapable.

4.2 Ambiguity

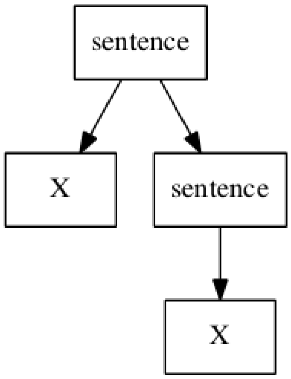
The processing of sentences in a language to get their meaning usually uses the parse trees. The parse tree is a way to understand the structure of what was said in the sentence. If for a given grammar G there exists a sentence in the language for which there is more than one parse tree then G is said to be **ambiguous**. You only need one example sentence to make G ambiguous! Note that the language is not ambiguous, the grammar is. Also note that it is OK for there to be more than one derivation for a sentence using an unambiguous grammar. Derivation doesn't make the grammar ambiguous, parse trees do.

```
<sentence>  ::= X | <sentence> X | X <sentence>
```

This grammar is ambiguous because there exists a sentence that has more than one parse tree. For example, XX has two parse trees:



and



Here is another way to get ambiguity for that language:

```
<sentence>  ::= X | <sentence> <sentence>
```

Try the sentence `XXX` on above grammar. Can you can find two parse trees? The fix is just to do:

```
<sentence>  ::= X | <sentence> X
```

Here is another very similar ambiguous grammar:

```
<binary-string>  ::= 0 | 1 | <binary-string> <binary-string>
```

Here is a fix:

```
<binary-string>  ::= 0 <binary-string> | 1 <binary-string> | 0 | 1
```

Ambiguity effects meaning:

```
<sentence>                                             ::= <expression>
<expression>                                           ::= <expression> + <expression> |
<expression> * <expression> | <identifier> <identifier>  ::= X | Y | Z
```

These are in the language:

```
X X + Y X + Y * Z
```

While the first two expressions are fine, `X + Y * Z` has a multiple parse trees. We want the multiply to be done first when we do a depth first traversal of the parse tree to extract the structure. How do we do this? We need to control operator precedence. Here we have added to the ambiguous grammar above to make multiply done before addition (BUT IT IS STILL AMBIGUOUS)

```
<sentence>    ::= <expression>
<expression>  ::= <term> | <expression> + <expression>
<term>        ::= <identifier> | <term> * <term>
<identifier>  ::= X | Y | Z
```

4.2.1 Ambiguity by Ill-specified Associativity

Operator Associativity determines the order of execution of homogeneous operators. Essentially is it left to right or right to left. Here we have modified the grammar to be left to to right for both + and *.

```
<sentence>    ::= <expression>
<expression>  ::= <term> | <expression> + <term>
<term>        ::= <identifier> | <term> * <identifier>
<identifier>  ::= X | Y | Z
```

Suppose we want to have both + and - be of equal precedence? We can do that here:

```
<sentence>    ::= <expression>
<expression>  ::= <term> | <expression> + <term> | <expression> - <term>
<term>        ::= <identifier> | <term> * <identifier>
<identifier>  ::= X | Y | Z
```

However, we cannot have two operators of the same precedence in which one is left to right associative and the other is right to left associative. Here is a simplified version of the problem:

```
<sentence>    ::= <expression>
<expression>  ::= <term> | <expression> + <term> | <term> - <expression>
<term>        ::= X | Y | Z
```

If we did then the grammar would be ambiguous. The above grammar has + is left to right as normal but - is right to left. How do we parse: X-Y+Z? It could be either (X-Y)+Z if the expression is expanded on the lhs of the + production or it could be X-(Y+Z) if the expression is expanded on the rhs of the - production. Furthermore, consider X+Y-Z. Only the lhs of the + can be expanded, so X+Y-Z is not even in the language!

Interestingly, the Dangling Else Problem (see below) is essentially ambiguity through ill-defined associativity as in the first case.

Finally, the grammar of the **Perl language** provides an interesting insight into associativity problems. In Perl, the language allows for two kinds of if statements. The first is **if (test) { statement }** and the second is **statement if (test)** . The grammar or Perl seems to inexplicably require braces in the first case. But suppose the grammar for Perl had been designed so braces are not required in the first case. What would happen? A little thought shows that the grammar without the braces is ambiguous. Let's walk through it. Assume the new grammar allows:

```
<stmt>  ::= if ( <test> ) <stmt> | <stmt> if ( <test> )
```

Then how would an intermediate statement of the form:

```
if ( <test> ) <stmt> if ( <test> )
```

be parsed? Would it be the same as:

```
if ( <test> ) <stmt>  if ( <test> )
```

or would it be equivalent to :

```
if ( <test> )  <stmt> if ( <test> )
```

4.2.2 Ambiguity by Infinite Loop

Why just adding stuff can be a bad idea: (infinite loop). We added a reference "up the grammar" to an earlier nonterminal. The result in a loop of `<expression>` → `<term>` → `<expression>` → `<term>` → ... that can go on any number of times. Therefore there is more than one parse tree since you can pick to go through the loop say 12 times or 317 times giving two different trees.

```
<sentence>    ::= <expression>
<expression>  ::= <term> | <expression> + <term>
<term>        ::= <identifier> | <term> * <identifier> | <expression>
<identifier>  ::= X | Y | Z
```

The fix is to mark each time through the loop by forcing something to be "removed" from the sentence each time. In this case we have parentheses:

```
<sentence>    ::= <expression>
<expression>  ::= <term> | <expression> + <term>
<term>        ::= <identifier> | <term> * <identifier> | ( <expression> )
<identifier>  ::= X | Y | Z
```

Now if you go through the loop 12 times you are nested 12 deep in parens but if you go through it 317 times then you are nested 317 deep.

4.2.3 Ambiguity by Multiple Paths

A grammar is ambiguous if you have two or more paths through the grammar to a symbol. For example starting with `<aardvark>`:

```
<aardvark>  ::= <cat> | <dog>
<cat>       ::= <zebra>
<dog>       ::= <zebra>
```

then you get at least two parse trees for `<zebra>` derived from `<aardvark>`. So this grammar is ambiguous. Note that

```
<aardvark>  ::= ( <cat> ) | <dog>
<cat>       ::= <zebra>
<dog>       ::= <zebra>
```

is NOT ambiguous since `<zebra>` must go through `<dog>` or it will pickup a set of parentheses "marking" that it went via `<cat>`.

In this next grammar just adding on stuff to a working grammar has created two ways to get from expression to X making the grammar ambiguous.

```
<expression>  ::= <term> | <expression> + <term> | <identifier>
<term>        ::= <identifier> | <term> * <identifier>
<identifier>  ::= X | Y | Z | <expression>
```

The first parse tree is:

```
<expression> -> <term> -> <identifier> -> X
```

The second is:

```
<expression> -> <identifier> -> X
```

In summary, there are three ways to get ambiguity.

- associativity:

```
<sentence>  ::= X | <sentence> <sentence>
```

- infinite loop:

```
<expression>  ::= <term> | <expression> + <term>
<term>        ::= <identifier> | <term> * <identifier>
<identifier>  ::= X | Y | Z | <expression>
```

- alternate routes:

```
<expression>  ::= <term> | <expression> + <term> | <identifier>
<term>        ::= <identifier> | <term> * <identifier>
<identifier>  ::= X | Y | Z | <expression>
```

5 The Dangling Else

Here is the **grammar for the dangling else** problem. In this problem we want to make sure that we always associate an `else` with last `if`. This prevents the ambiguity of not knowing which `if` an `else` associates with. Let's work through this carefully.

Suppose you have the grammar:

```
stmt :  IF '(' exp ')' stmt ELSE stmt
| IF '(' exp ')' stmt
```

Note the ambiguity with the grammar this is exposed with this statement:

```
if ( exp ) if ( exp ) stmt else stmt
```

Is the `else` associated with the first `if` or the second? You can't tell from the grammar alone! We want the grammar to force the `else` to be associated with the second `if`!! It is the rule of the language and what most languages do called the rule of "most recent if".

One way to solve this problem is to force all if statements to have an `endif`. This is forces the user to choose which interpretation of the statement they want the `else` matched with the second `if`:

```
if ( exp ) if ( exp ) stmt else stmt endif endif
```

or the first:

```
if ( exp ) if ( exp ) stmt endif else stmt endif
```

5.1 The Matched/Unmatched Solution

Let's now solve this my creating separate nonterminals for if-statements with a paired `else` and without. Let a "matched" if be one in which an `else` is matched with the the most recent unmatched `if`. Let an "unmatched" if be an `if` with no matching `else`. This gives us 6 possible cases for an if-statement whose then-part or else-part may also be an if-statement. I have labeled each with a number and whether the statement ends with an unmatched `if` or not. Here are the 6 cases:

```
IF '(' <exp> ')' <matched>                         // 1 unmatched
IF '(' <exp> ')' <unmatched>                       // 2 unmatched
IF '(' <exp> ')' <matched> ELSE <matched>          // 3 matched
IF '(' <exp> ')' <matched> ELSE <unmatched>        // 4 unmatched
IF '(' <exp> ')' <unmatched> ELSE <matched>        // 5 matched
IF '(' <exp> ')' <unmatched> ELSE <unmatched>      // 6 unmatched
```

To have our rule of "the most recent unmatched if matches the else" we must discard rules 5 and 6 because they have an unmatched `if` before the `else`. So using 1 through 4 above gives us a grammar like:

```
<stmts>      ::= <stmts> <stmt> | <stmt>
<stmt>       ::= <matched> | <unmatched>
<matched>    ::= IF '(' <exp> ')' <matched> ELSE <matched>
             | <other-stmts>
<unmatched>  ::= IF '(' <exp> ')' <matched>
             | IF '(' <exp> ')' <unmatched>
             | IF '(' <exp> ')' <matched> ELSE <unmatched>
```

Note that `<other-stmts>` is in the "matched" class because every `if` is matched by an `else` because there are no `if`s. Also note that the `if` only occurs in the matched and unmatched cases and nowhere else.

Here is a common mistake with the **dangling else** grammar:

```
<stmts>      ::= <stmts> <stmt> | <stmt>
<stmt>       ::= <matched> | <unmatched> | <other-stmts>
<matched>    ::= IF '(' <exp> ')' <matched> ELSE <matched>
             | <other-stmts>
<unmatched>  ::= IF '(' <exp> ')' <matched>
             | IF '(' <exp> ')' <unmatched>
             | IF '(' <exp> ')' <matched> ELSE <unmatched>
```

In this case the grammar is ambiguous since there is more than one way to get to `<other-stmts>`! The correct way to get `<other-stmts>` is by: `<stmts>` → `<matched>` → `<other-stmts>`

Here is another common mistake with the dangling else grammar:

```
<stmts>      ::= <stmts> <stmt> | <stmt>
<stmt>       ::= <matched> | <unmatched> | <other-stmts>
<matched>    ::= IF '(' <exp> ')' <matched> ELSE <matched>
<unmatched>  ::= IF '(' <exp> ')' <matched>
             | IF '(' <exp> ')' <unmatched>
             | IF '(' <exp> ')' <matched> ELSE <unmatched>
```

Not only is this dangling else grammar incomplete it has a trap in it. Suppose you get to `<matched>` or `<unmatched>` then you can never escape those two states. And if you can't escape you are stuck either matching if statements forever or you error when you come across more legal code. Therefore it is incomplete in that sense.

5.2 Mixing in the While Statement

That's the simple Dangling Else Problem. But wait! Suppose the grammar you are looking at has something like:

```
stmt :  WHILE '(' exp ')' stmt
| IF '(' exp ')' stmt ELSE stmt
| IF '(' exp ')' stmt
```

The problem is complicated by the fact that both `if` and `while` statements do not have any end of statement marker they can be intermingled leading to ambiguity.

```
if ( exp ) while (exp) if ( exp ) while (exp) stmt else stmt
```

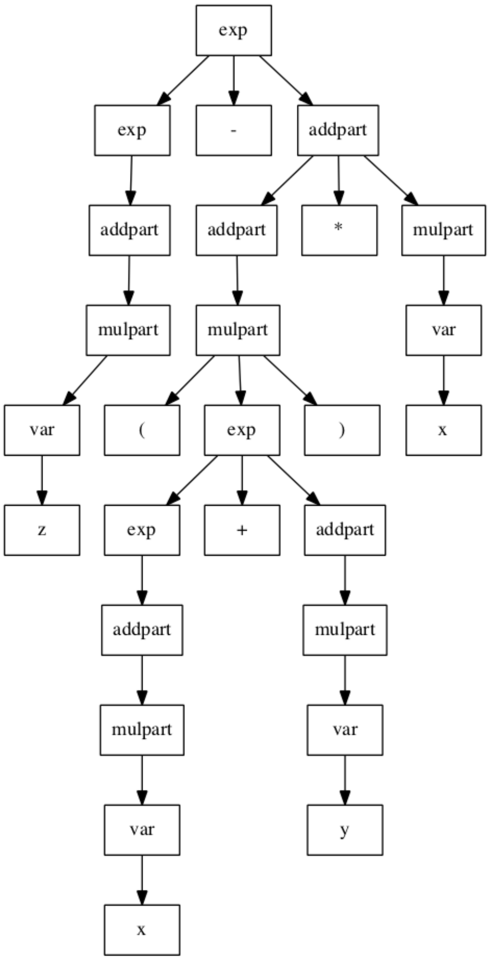
To get this to work with our matched/unmatched scheme we have to mix in the following with the subset of 6 cases of matched/unmatched for the `if`. That is, `if` and `while` must be mixed together in the same matched/unmatched productions:

```
WHILE '(' <exp> ')' <matched>    // 1 matched
WHILE '(' <exp> ')' <unmatched>  // 2 unmatched
```

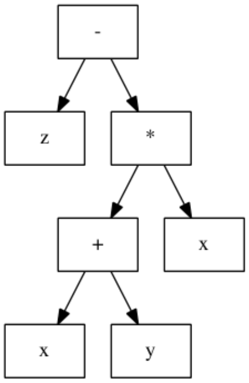
This is essentially splitting the `while` into two classes: the `while`s containing statements with unmatched `if`s, and the `while`s with matched ones. If you have other loops ending in statement with no end loop marker you need to do the same for all of them.

6 The Difference between an AST and a Parse Tree

An **Abstract Syntax Tree (AST)** is the representation of the meaning of the statement in tree form. It involves the necessary and sufficient nodes to cover that operators and operands of any statement in the language. The Parse Tree represents the substitutions made based on a grammar in evaluating whether a statement is in the language or not. The AST for in if statement like `if (x>y) z=10;` does not contain the parentheses for instance. It has a node for an `if` and the `test` and the `assignment`. Here is the parse tree and abstract syntax tree for the statement $z - (x + y) * x$:



Parse Tree



AST

7 The Power of Context Free and incomplete grammars

Another problem with grammars is they do not describe all the sentences of language you had in mind. Let's look at that along with the idea that a CFG is context free.

In a context free language you can make any substitution for a left-hand-side nonterminal suggested by any production without regard to the context in which the nonterminal occurs. For example

```
<dogs>  ::= corgi | other
```

means that **anywhere** a dog nonterminal occurs you can replace it with a corgi or an other. Let's use this mathematical property of grammars to see if there is a difference between two languages specified by these two grammars.

What is the difference between L_a and L_b?

L_a :

```
<sentence>  ::= <exp>
<exp>       ::= <term> | <exp> + <term>
<term>      ::= <id> | <term> * <id>
<id>        ::= X | Y | Z | ( <exp> )
```

L_b :

```
<sentence>  ::= <exp>
<exp>       ::= <term> | <exp> + <term>
<term>      ::= <id> | <term> * <id> | ( <exp> )
<id>        ::= X | Y | Z
```

Note that by using the rule of context-freeness we can transform the grammar of L_a to:

```
<sentence>  ::= <exp>
<exp>       ::= <term> | <exp> + <term>
<term>      ::= <id> | <term> * <id> | (<exp>) | <term> * (<exp>)
<id>        ::= X | Y | Z
```

so this adds `<term> * (<exp>)` that wasn't in L_b. What is an example of this:

```
X*(Y)
```

Therefore `X*(Y)` is in L_a but not in L_b!!!

8 Tips for Writing Successful Grammars

- There must be one of the alternations in a production that is not recursive. For example:

  ```
  <list>  ::= <list> dogs | "(" <list> ")" | if <list> then dogs | my <list>
  ```

  can never get rid of the `<list>` nonterminal since any replacement has a `<list>` in it. A fix would be change the language:

  ```
  <list>  ::= <list> dogs | "(" <list> ")" | if <list> then dogs | <other>
  ```

  or

  ```
  <list>  ::= <list> dogs | "(" <list> ")" | if <list> then dogs | cats
  ```

- Try to build your grammar from top to bottom. Each production only referring to nonterminals farther down the page. If you do refer to a production up the page then make sure you remove a token when you return up the tree such as in the parenthesize expression example above. Every time you go up you will have to take a token off the input so the process will have to finish.

- Build hierarchically. Each production can be used to handle a specific case. For example: Here is a number:

  ```
  <num>    ::= <digit> <num> | <digit>
  <digit>  ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
  ```

  Here we add a production for the signed number:

```
<signed>  ::= + <num> | - <num> | <num> <--- added
<num>     ::= <digit> <num> | <digit>
<digit>   ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Here we add a production for putting a nonzero digit at the beginning. How was 0 handled?

```
<signed>   ::= + <num> | - <nznum> | <nznum>
<nznum>    ::= <nzdigit> <num> | <digit> <--- added
<num>      ::= <digit> <num> | <digit>
<digit>    ::= 0 | <nzdigit>
<nzdigit>  ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

9 Extended BNF

Extended BNF tries to make lists and optional arguments easy. Extended BNF uses metachars [], , (), *, +

- optional elements (0 or 1 times) are denoted []
- grouping of alternatives are denoted ()
- sequences {}`*` means zero or more times (the star is called a **Kleene Star**)
- sequences {}`+` means one or more times

Note: sometimes {} means {}`*`. EBNF often says nothing about precedence or associativity and this is often defined with text about the operators. Here are a bunch of examples:

- {`X`} is zero or more X's
- `if <exp> then <stmt> [else <stmt> ]` is a use of the optional designator.
- {`<stmt>`}`+` is a list of one or more statements
- `XX` is another way to do 1 or more
- `X` { `, X` } is a comma separated list of X's
- `<num>` { `(+|-) <num>` } This allows just a single number or an expression consisting of numbers that are added and subtracted.
- { `X ;` } is the case of a list of semicolon terminated `X`'s