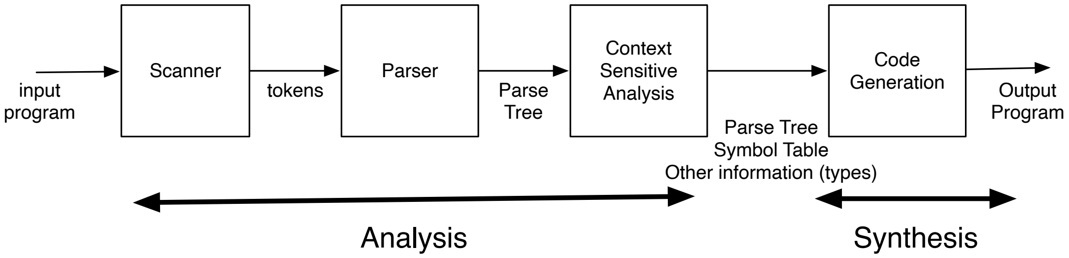
Compiled on 14/09/2022 at 5:38am

Context-Free Grammars

This module will cover Context-Free Grammars (CFGs). But, before we talk about CFGs, it is worth discussing how Context-Free Grammars relate to our course objectives. As discussed earlier, the course is about abstractions; we intend to show different levels of abstractions by using a higher-level language (a small subset of C) and discussing everything that is needed to move from this higher level all the way down to machine language.

The assembler discussed in the Machine Language Module was a translator that moved from Assembly language to Machine Language. Since the abstraction jump between Assembly and Machine language is small, writing the assembler was relatively straightforward. The jump between a higher-level language to Assembly is bigger, and therefore the translator/compiler is more complicated.

The figure below shows the main components of a typical compiler.



In the Regular Languages module, we discussed how to write a Scanner for a language. Recall that the result of scanning is a sequence of tokens. The next step in the compilation process takes the sequence of tokens and checks for syntax errors. In doing so, the *Parser* creates a representation of the program (called a *Parse Tree*). It is because of this that the Parser is often referred to as the syntactic analysis stage. However, some requirements in high-level languages go beyond syntax (e.g., types in an assignment statement must match, a variable must have been declared before being used). These requirements must also be confirmed to determine if the input program is a valid program. Parsing algorithms, being unaware of such contextual information, are unable to perform these checks. The next stage in compilation, *Context-Sensitive Analysis*, takes care of this (this is often also referred to as the *Semantic Analysis* stage). Finally, once the compiler has confirmed that the program is indeed valid, the compiler moves to the *Synthesis* stage and generates the intended output.

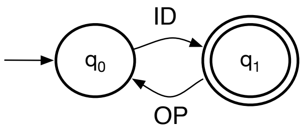
An alternate view of thinking about the different stages of compilation is to think of each analysis stage as weeding out invalid input programs. The scanner weeds out programs that are lexically invalid, i.e., contain invalid characters or character combinations, and thus cannot be split into tokens. Similarly, only those programs which are syntactically valid move on to the Context-Sensitive Analysis stage. Finally, only programs that are lexically, syntactically and semantically valid move to the Code Generation stage. Typically, a program is defined as “valid” (e.g., a “valid C program”) if it passes these phases but note that it could still be incorrect with respect to the programmer’s intent (i.e., it could still have bugs). There are many different ways that a program may be correct or incorrect and being able to distinguish them will make you a more effective programmer.

1 The need for Context-Free Grammars

Our next stage in compilation is the parsing stage and it requires us to learn about Context-Free Languages (specified using Context-Free Grammars). The obvious question to ask is why? In other words, why is it not possible to specify the syntax of our language using a regular language? Could we not specify a regular language where tokens are the alphabet and words in the language are valid statements/instruction in the language whose syntax we are checking? The short answer is that regular languages are too restrictive for representing the complex structure of higher-level languages. For a longer answer, we use the following motivating example for a simple language whose syntax we want to validate:

$\Sigma =$ {ID, OP} and $L = \{w : w$ valid arithmetic expressions$\}$

Notice that ID and OP represent tokens (ID tokens would be variable names, OP would be operators such as plus or minus). Valid arithmetic expressions would be words such as ID or ID OP ID. The following DFA represents a regular language which will accept words that are valid arithmetic expressions:

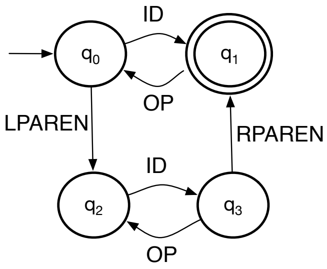


One issue with this language is that it does not enforce ordering, i.e., there is no way to specify the order in which the operations should be applied. We can try to extend the language by allowing parentheses:

$\Sigma =$ {ID, OP, LPAREN, RPAREN} and $L = \{w : w$ valid arithmetic expressions$\}$

LPAREN is a token indicating a left parenthesis, (, and RPAREN is a token indicating a right parenthesis,). Using this extension, we could write operations such as ID OP LPAREN ID OP ID RPAREN to represent input such as `a-(b*c)`.

The following DFA will recognize expressions that never nest parentheses. For example, expressions such as `a+b`, `(a+b)*c` and `(a+b)*(c-d)` will be recognized as valid words in the language.



Could we extend the DFA to allow two levels of nested parentheses? What about three levels? [1]

[1] The DFA can be extended to allow for any finite level of nesting by simply repeating what we did to add the first level of parentheses.

But that is still restrictive; we would like to arbitrarily nest parentheses! This is impossible with a regular language since a DFA by definition must have a finite (the F in DFA) number of states. The challenge is that once you have arbitrarily large number of open parentheses, you need to ensure that the same number of close parentheses are matched. In a DFA, for each pair of open and close parentheses, you need two extra states to keep a count of how deeply nested the expression is. This example is called the *Balanced Parentheses Problem* and is the classic example to demonstrate the limitations of regular languages.

This example hints at a deeper fact regarding regular languages. Regular languages are, fundamentally, the class of languages that can be recognized by a computer with a finite amount of memory. This might not seem like a problem, because all computers of course only have a finite amount of memory—indeed, for a computer to exist in the physical universe, it *must* have a finite amount of memory. But from a theoretical viewpoint, this is a serious limitation. Recognizing structures with an arbitrary amount of nesting, like parentheses nested within each other, or nested conditional statements and loops in a programming language, cannot be done with a finite amount of memory. At the very least, you need a counter to keep track of how deep the nesting level is, and since this counter can get arbitrarily large, it cannot be contained in a finite amount of memory.

So, if real computers, with their finite amount of memory, can only recognize regular languages, how do they deal with the nested structures in programming languages? The obvious answer is that they only accept programs that don't exceed a certain level of nesting. What's interesting is that this limit doesn't need to be hard-coded. Rather than saying explicitly that only 1000 levels of nesting are allowed, a compiler can simply keep using the memory allotted to it by the operating system until it runs out. Effectively, the compiler is pretending that it has an **infinite amount of memory**, and freely using as much as it needs until it bumps into the limit. When you write programs, you probably do the same: rather than explicitly allocating an array of size 1000, you create a vector or list that can grow as much as it needs. Keeping this common programming practice in mind, it doesn't seem so strange to move beyond finite memory and discuss new kinds of languages which require infinite memory to recognize.

In the following section, we discuss Context-Free Languages, which have the expressive power to represent, among other things, structures with arbitrary nesting. Though we will not discuss the model of computation corresponding to Context-Free Languages (called a Pushdown Automaton), the key feature of this model is that it has access to an infinitely large stack, allowing it to overcome the limits of finite-memory computation.

2 Context-Free Grammars

A regular language could be expressed by a DFA, NFA, ϵ-NFA or regular expression. Of those, the regular expression is the most "human", in that it's fairly straightforward for a human to use to express a regular language, and the DFA is the most "computer", in that it's quite efficient for a computer to use. The next modules will focus on parsing; in this module, we focus on the expression of Context-Free Languages, and so need a context-free equivalent to the regular expression.

A *grammar* is the language of languages (Matt Might). A grammar helps us describe what we are allowed and not allowed to say. In fact, a regular expression can also be called a regular grammar, although calling it so is rare in practice. A context-free language is typically expressed as a *context-free grammar*.

The easiest way to understand context-free grammars is to start from regular expressions and fix the limitations they had. If we wanted to write a regular expression to solve the Balanced Parentheses Problem, we would start with something like

$$\texttt{(}\,???\,\texttt{)}\,|\,\epsilon$$

The "*???*" here represents the problem: what we would like is to say that what we can nest in the matched parentheses is... more balanced parentheses. We could solve this problem if we could put the entire language in itself, i.e., recurse within a regular expression. That's exactly what context-free languages add, in terms of power, to regular languages: recursion. We could instead imagine describing this language like so:

$$L = \texttt{(}L\texttt{)}\,|\,\epsilon$$

The equals sign is a bit questionable in the above language, as there's no way to simplify this as an expression, so instead, we describe it as a rewriting process, with the symbol →, and repeatedly rewrite the L part into one of its expanded forms to reach a string in the language. Expounding on this idea and dropping some features of regular expressions that can be replicated with recursion results in context-free grammars.

We will discuss the meaning of defining languages in terms of rewriting rules after formally defining CFGs:

Definition 1 *A **Context-Free Grammar (CFG)** is a 4-tuple (N, T, P, S) where*

- *N is a finite, non-empty set of **non-terminal symbols**.*
- *T is an alphabet; a finite, non-empty set of **terminal symbols**. (Some use Σ rather than T)*
- *P is a finite set of **productions/rules**, each of the form $A \to \beta$ where $A \in N$ and $\beta \in (N \cup T)^*$.*
- *$S \in N$ is a **start symbol**.*

*Note: We write $V = N \cup T$; this is called the **vocabulary**, the set of all symbols in our language.*

As a concrete example, the following CFG represents valid arithmetic expressions with arbitrary balanced parentheses:

```
N = { expr }
T = { ID, OP, LPAREN, RPAREN }
Productions:
expr → ID
expr → expr OP expr
expr → LPAREN expr RPAREN
S = expr
```

Notice that terminals never occur on the left-hand side (LHS) of a production rule. The above grammar has three productions (or rules) and all of them have the non-terminal `expr` on the LHS. It is important to also notice that each left-hand side of a rule must have a single non-terminal. Terminals and non-terminals can both appear on the right-hand side (RHS) of a rule. We discuss the need to differentiate between non-terminals and terminals again in the Derivations section below.

2.1 Conventions

We use some conventions which allow us to state formal definitions concisely without having to repeatedly specify which set each variable belongs to.

- Lower-case letters from the start of the alphabet, i.e., $a, b, c, ...$, are elements of T (*terminals*)
- Lower-case letters from the end of the alphabet, i.e., w, x, y, z, are elements of T^* (*words*)
- Upper-case letters from the start of the alphabet, i.e., $A, B, C, ...$, are elements of N (*non-terminals*)
- Greek letters, i.e., $\alpha, \beta, \gamma, ...$, are elements of V^* (recall this is $(N \cup T)^*$, *sequences of terminals and non-terminals*)
- When not specified formally, the non-terminal on the left-hand side (LHS) of the first production is the start symbol.

2.2 Derivations

Definition 2 *Over a CFG (N, T, P, S), we say that A directly derives γ, and write $A \Rightarrow \gamma$, if and only if there is a rule $A \rightarrow \gamma$ in P.*

In other words, the "directly derives" relation represents a single application of a production rule. Informally, the idea is to **rewrite** (or expand) a non-terminal by replacing the left-hand side (LHS) of the production rule with the right-hand side (RHS).

Given an arbitrary sequence $\alpha A \beta$, we use the definition of "directly derives" to say that $\alpha A \beta \Rightarrow \alpha \gamma \beta$ if and only if $A \rightarrow \gamma$.

Definition 3 *Over a CFG (N, T, P, S), we say that α derives β, and write $\alpha \Rightarrow^* \beta$, if either $\alpha = \beta$, or if there exists γ such that $\alpha \Rightarrow \gamma$ and $\gamma \Rightarrow^* \beta$.*

In other words, we derive the sequence β from α through zero or more applications of production rules. For the mathematically inclined, this is the reflexive transitive closure of the "directly derives" relation.

Definition 4 *Over a CFG (N, T, P, S), a derivation of a string of terminals x is a sequence $\alpha_0\alpha_1 \cdots \alpha_n$ such that $\alpha_0 = S$ and $\alpha_n = x$ and $\alpha_i \Rightarrow \alpha_{i+1}$ for $0 \leq i < n$.*

Example: For the grammar of expressions with balanced parentheses, give a derivation for the word ID OP LPAREN ID OP ID RPAREN, i.e., show `expr` $\Rightarrow^*$ `ID OP LPAREN ID OP ID RPAREN`.

Recall that a derivation begins with the start symbol, which is `expr` for our grammar.

```
expr ⇒ expr OP expr                     (applying expr→expr OP expr)
     ⇒ ID OP expr                       (applying expr→ID)
     ⇒ ID OP LPAREN expr RPAREN         (applying expr→LPAREN expr RPAREN)
     ⇒ ID OP LPAREN expr OP expr RPAREN (applying expr→expr OP expr)
     ⇒ ID OP LPAREN ID OP expr RPAREN   (applying expr→ID)
     ⇒ ID OP LPAREN ID OP ID RPAREN     (applying expr→ID)
```

Each step of the derivation simply chooses a non-terminal from the current α_i and rewrites it by replacing it with the RHS of some rule for that non-terminal. This is what we meant when we said earlier that context-free grammars are a set of rewrite rules. When we formally defined context-free grammars, we mentioned that we would address the need to differentiate between terminals and non-terminals. Recall that only non-terminals appear on the left-hand side of rules. This is because non-terminals do not *terminate*; they can be expanded by applying a rule which replaces the left-hand side with the right-hand side. Additionally, since each rule must have only one non-terminal on the left-hand side, that rule can be applied whenever desired. These grammars are named Context-Free because of this; regardless of the context, i.e., not depending on any other symbol, the left-hand side of a rule (a single non-terminal) can be replaced by its right-hand side. The other symbols, terminals, are terminals since they terminate, i.e., they can no longer be expanded since they never appear on the left-hand side of a rule.

2.3 Context-Free Languages

Definition 5 *The language of a CFG (N, T, P, S) is $L(G) = \{w \in T^* : S \Rightarrow^* w\}$.*

In other words, a context-free language is the set of words (strings of terminals) that have a derivation in G, i.e., words can be derived starting at the start symbol of G. Notice that a string which contains one or more non-terminals cannot be a word in a context-free language; i.e., by definition, words in a language are strings of terminals.

Definition 6 *A language L is context-free if and only if there exists a CFG G such that $L = L(G)$.*

Every regular language is context free! In Module 3, we defined a language as regular if (1) it is the empty language or the language consisting of the empty word (2) the language {a} for each $a \in \Sigma$ and (3) the language built using the union, concatenation or Kleene star of two regular languages. It suffices to show that for each of these conditions it is possible to create a corresponding context-free grammar (and therefore a context-free language) that represents the regular language. We leave this as an exercise for the interested reader.

Example: Let $T = \{a, b\}$. Write a CFG for $\{a^n b^n : n \in \mathbb{N}\}$. Further, find a derivation in your grammar for the string *aaabbb*.

```
N = {S}
T = {a,b}
Productions:
S → ϵ
S → aSb
```

We will often use the shorthand $S \rightarrow \epsilon | aSb$ to compactly represent multiple productions for the same non-terminal. We will also often just write the production rules (since the other components of a CFG can be easily inferred from these rules). Recall also that when not explicitly stated, the non-terminal on the Left-Hand-Side of the first rule is the start symbol (S in this case).

Recall that a derivation begins at the start symbol and derives a string containing terminals. The following is a derivation for the string *aaabbb*: $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaabbb$.

Write a CFG for palindromes over $\{a, b, c\}$. Strings such as a, $abba$, $acbbca$ are words in the language defined by the grammar.

[2]

Write a CFG for the regular language defined by the regular expression $a(a|b)^*b$.

[3]

[2]
```
N = {S,M}
T = {a,b,c}
Productions:
S → aSa | bSb | cSc | M
M → a | b | c | ϵ
```
Notice that while we introduced an additional non-terminal M, it is not strictly needed, i.e., an alternate set of productions using just one non-terminal, S, could be created.

[3]
```
N = {S,M}
T = {a,b}
Productions:
S → aMb
M → aM | bM | ϵ
```

3 Optional Reading: Reduced Grammars

Consider the following grammar:

S $\rightarrow$ A | B
A $\rightarrow$ a
B $\rightarrow$ B b
C $\rightarrow$ c

Recall that words in the language are strings of terminals that can be derived starting at the start symbol. S directly derives A and B. A directly derives a while B directly derives Bb. What is interesting to note is that we can never obtain a partial derivation α which contains the non-terminal C. This means that there is no reason for having the non-terminal C and the rules associated with it. We say that this grammar is not a `reduced grammar`.

Suppose we removed the rule for C. There is another peculiar rule in this grammar. Any partial derivation that uses B will never actually lead to a derivation. That is because every application of the rule for B results in Bb. We say that this grammar is still not a `reduced grammar` since it gets stuck in a rewrite loop.

It is difficult to say what the intent of this grammar is. If the intent is to represent the regular language $a|b^*$, the corresponding CFG would be

S $\rightarrow$ A | B
A $\rightarrow$ a
B $\rightarrow$ B b | ϵ

In particular, notice how the recursion in the rule for B is terminated using ϵ. If the regular language we wanted to represent was $a|bb*$ ($a|b+$ if we were to use extended regular expressions), we would change the rules for B to be:

B $\rightarrow$ B b | b

A real-life example of this can be found in high-level programming languages. The following is an excerpt from the context-free grammar for the JOOS language (a subset of Java taught in CS444). The rules insist there be at least one “Modifier” for a method.

Modifier $\rightarrow$ public | protected | static | abstract | final | native
Modifiers $\rightarrow$ Modifiers Modifier | Modifier

In CS241, we only discuss reduced grammars, i.e., grammars that have no useless non-terminals and where all recursive rules terminate.

4 Derivations and parse trees

Recall that a recognition algorithm tells us whether a word is in a language. For the specific case of a compiler, we are interested in more than just knowing that the input program is syntactically correct (i.e., in the language). We would also like to obtain a representation of the structure of the program. The derivation obtained to confirm that a word is indeed in the language can be used to provide such a representation.

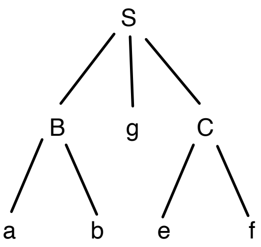
Consider this rather simple finite language represented as a CFG:

```
N = {S,B,C}
T = {a,b,c,d,e,f,g,h}
Productions:
S → BgC
B → ab|cd
C → h|ef
```

This grammar specifies the language $L(G) = \{$`abgh,abgef,cdgh,cdgef`$\}$. We know that `abgef` $\in L(G)$ since we can provide the following two derivations for the word:

```
S ⇒ BgC ⇒ Bgef ⇒ abgef
S ⇒ BgC ⇒ abgC ⇒ abgef
```

Using the derivations, we can also visually represent the structure of the input word. This representation is called a `parse tree`. The grammar was chosen carefully such that either of the derivations above yield the same parse tree:



The root of the parse tree is the start symbol. Each non-leaf node is a non-terminal, and its immediate descendants are the right-hand side of the rule that was used for the non-terminal in the derivation. For example, C is a non-terminal, and its two descendants `e` and `f` form the right-hand side of the rule `C` $\rightarrow$ `ef`. Leaf nodes represent terminals or ϵ. It's common to exclude the ϵ leaf node for ϵ productions, in which case a non-terminal can also be a leaf node, but only if its expansion was ϵ.

Based on how parse trees are created, we have the following properties:

Property 1 *A derivation uniquely defines a parse tree.*

Property 2 *An input string can have more than one parse tree.*

While it is easy to convince ourselves that Property 1 holds, it is worth looking more into Property 2. In the example above, even though we had two different derivations, they led to the same parse tree. However, Property 2 tells us that this is not always true.

Can you come up with a CFG and an input string such that the string has two different parse trees with respect to the CFG?

[4]

Since the parse tree representation within a compiler is meant to represent the structure of a program, it is important that the compiler produces unique parse trees. The reason the input string `abgef` yielded two derivations is because at each stage of the derivation we had a choice of which non-terminal to `expand` (by applying a production rule for the non-terminal). We can eliminate this uncertainty as follows:

[4] We discuss a grammar that yields different parse trees because it yields multiple derivations for the same input string in the next section.

Definition 7 *In a Leftmost Derivation, alternately Left Canonical Derivation, we always expand the leftmost non-terminal first. Formally, each step has the form:*

$$xA\gamma \Rightarrow x\alpha\gamma$$

In a Rightmost Derivation, alternately Right Canonical Derivation, we always expand the rightmost non-terminal first, Formally, each step has the form:

$$\beta Ax \Rightarrow \beta\alpha x$$

Recall that x represents a string of terminals and α, β and γ are elements of V^*, words over the vocabulary.

The derivation `S` $\Rightarrow$ `BgC` $\Rightarrow$ `Bgef` $\Rightarrow$ `abgef` is a Rightmost Derivation since it always chooses to apply a production rule, i.e. expand, the rightmost non-terminal. Alternately, the derivation `S` $\Rightarrow$ `BgC` $\Rightarrow$ `abgC` $\Rightarrow$ `abgef` always expands the leftmost non-terminal and is therefore a Leftmost Derivation. Notice that for this grammar, if we were to choose one derivation style, we would yield a unique derivation for the word `abgef`.

5 Ambiguous Grammars

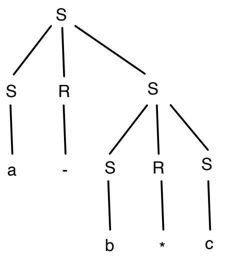
Consider the following CFG for arithmetic operations which is similar in structure to the first CFG we saw in this module:

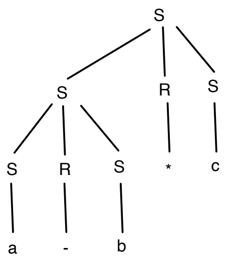
$$\begin{array}{lcl} \text{S} & \rightarrow & \text{a} \,|\, \text{b} \,|\, \text{c} \,|\, \text{SRS} \\ \text{R} & \rightarrow & + \,|\, - \,|\, * \,|\, / \end{array}$$

Let's obtain a derivation for the input string `a - b * c`. Since we just discussed the importance of obtaining unique parse trees, we will choose to produce Leftmost derivations (we could have chosen Rightmost). It turns out that even if we select a particular derivation style, we can obtain two different derivations and in fact two different parse trees.

$$\begin{array}{l} S \Rightarrow SRS \Rightarrow aRS \Rightarrow a - S \Rightarrow a - SRS \\ \quad \Rightarrow a - bRS \Rightarrow a - b * S \Rightarrow a - b * c \end{array}$$

$$\begin{array}{l} S \Rightarrow SRS \Rightarrow SRSRS \Rightarrow aRSRS \Rightarrow a - SRS \\ \quad \Rightarrow a - bRS \Rightarrow a - b * S \Rightarrow a - b * c \end{array}$$





Definition 8 *A grammar is ambiguous if there is a word in the language which has more than one distinct leftmost derivation, or more than one distinct rightmost derivation.*

The grammar for arithmetic expressions discussed above is ambiguous; we obtained two distinct derivations of our input string **even though we chose the same derivation style**. If we cared only about recognition, i.e., determining whether $w \in L(G)$, then it wouldn't matter, since any derivation proves that the word is in the language. But two distinct leftmost derivations can only be distinct if they chose different expansions of some non-terminal, since they always choose the same non-terminal to expand (the leftmost one). So, by definition, two distinct leftmost derivations will have distinct parse trees. The same logic of course applies to rightmost derivations as well. Since parse trees give meaning to the string with respect to the grammar, we want to obtain unique parse trees. For example, parse trees representing arithmetic expressions can be evaluated[5] using a depth first post-order traversal. If we were to evaluate the tree on the left above, we would evaluate the node a, then node - and then the subtree on the right. The evaluation of this subtree first evaluates b, then $*$ then c. Then it computes the result $b * c$. With this subtree evaluated, the post order traversal computes the result `a-(b*c)`. Similarly, the tree above to the right first evaluates a, then -, then b and then the Root of this subtree, i.e., it computed a-b. It then evaluates $*$, then c and finally the Root of the entire tree to produce the result `(a-b)*c`. Since these are different values, our chosen parse trees have given different meanings to the same expression!

One way to avoid the ambiguity is to use precedence heuristics to guide the derivation process. For example, we could force the language syntax to require parentheses. The following grammar is unambiguous since it only accepts words such as `(a-(b*c))` or `((a-b)*c)`.

$S \; \text{–>} \; a \mid b \mid c \mid (SRS)$
$R \; \text{–>} \; + \mid - \mid * \mid /$

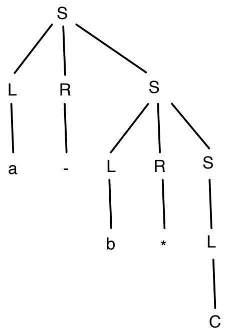
Give the unique leftmost derivations for `(a-(b*c))` and `((a-b)*c)`.

[6]

While this removes the ambiguity, imagine having to always use parentheses when writing arithmetic expressions in a programming language. Another way to remove ambiguity that does not require parentheses is to examine how parse trees work. As discussed earlier, in a parse tree, you evaluate the expression in a depth-first, post-order traversal. At the same time, we also want to interpret expressions such as `a-b+c` as `(a-b)+c` and not as `a-(b+c)`, i.e., we want left associativity. We can make a grammar left/right associative by insisting on how the recursion works!

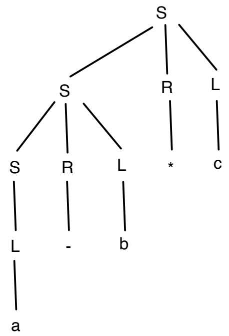
Right Associative Grammar

$S \rightarrow LR\mathbf{S} \mid L$
$L \rightarrow a \mid b \mid c$
$R \rightarrow + \mid - \mid * \mid /$



Left Associative Grammar

$S \rightarrow \mathbf{S}RL \mid L$
$L \rightarrow a \mid b \mid c$
$R \rightarrow + \mid - \mid * \mid /$



[5] That is, the mathematical value of the expression can be determined. E-*valu*-ation is determining the value of an expression.

[6] $S \Rightarrow (SRS) \Rightarrow (aRS) \Rightarrow (a–S) \Rightarrow (a-(SRS)) \Rightarrow (a-(bRS)) \Rightarrow (a-(b*S)) \Rightarrow (a-(b*c))$
$S \Rightarrow (SRS) \Rightarrow ((SRS)RS) \Rightarrow ((aRS)RS) \Rightarrow ((a–S)RS) \Rightarrow ((a–b)RS) \Rightarrow ((a–b)*S) \Rightarrow ((a–b)*c)$

Notice how in the grammar that associates to the left, the recursion on the non-terminal S is on the left. We will discuss the effects of left-recursive vs. right-recursive grammars on parsing in the next module.

Using this technique to change the associativity of a grammar, we can create a grammar that follows BEDMAS rules more closely by making $*$ and / appear further down the tree. Recall, with our depth-first traversal, the deeper parts of the tree will be evaluated first therefore get a higher precedence:

$$\begin{aligned} \mathrm{S} &\rightarrow \mathrm{SPT}\,|\,\mathrm{T} \\ \mathrm{T} &\rightarrow \mathrm{TRF}\,|\,\mathrm{F} \\ \mathrm{F} &\rightarrow \mathrm{a}\,|\,\mathrm{b}\,|\,\mathrm{c}\,|\,(\,\mathrm{S}\,) \\ \mathrm{P} &\rightarrow +|- \\ \mathrm{R} &\rightarrow *\,|\,/ \end{aligned}$$

Find a leftmost derivation for $a - b * c$ and draw the corresponding parse tree.

[7]

Consider the bitwise NOT operator ˜ which is a unary operator (takes one argument instead of two). This operator should have higher precedence than all the binary operators. How can we add this operator to the above grammar in a way that ensures it has the correct precedence?

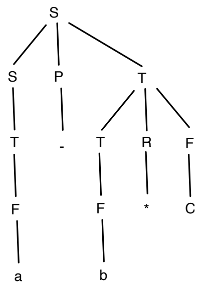
[8]

We have created a video that illustrates the connection between derivations, parse trees and how by changing the grammar we can change the meaning of a “program”. The video Post-Order Traversal is available on the course website.

Since + and - have the same precedence, and * and / have the same precedence, we will often use the following simplified expressions for the grammar for arithmetic expressions (even though it only allows + and * operations). This is essentially a convenience to keep the grammar and the derivations shorter.

$$\begin{aligned} \mathrm{S} &\rightarrow \mathrm{S} + \mathrm{T} \;\;|\;\; \mathrm{T} \\ \mathrm{T} &\rightarrow \mathrm{T} * \mathrm{F} \;\;|\;\; \mathrm{F} \\ \mathrm{F} &\rightarrow \mathrm{a}\,|\,\mathrm{b}\,|\,\mathrm{c}\,|\,(\,\mathrm{S}\,) \end{aligned}$$

[7]
$$\begin{aligned} \mathrm{S} &\Rightarrow \mathrm{SPT} \Rightarrow \mathrm{TPT} \Rightarrow \mathrm{FPT} \Rightarrow \mathrm{aPT} \\ &\Rightarrow \mathrm{a{-}T} \Rightarrow \mathrm{a{-}TRF} \Rightarrow \mathrm{a{-}FRF} \\ &\Rightarrow \mathrm{a{-}bRF} \Rightarrow \mathrm{a{-}b{*}F} \Rightarrow \mathrm{a{-}b{*}c} \end{aligned}$$



[8] A simple way is to add the rule F → ˜F. This ensures the NOT operator will appear deeper in the tree than the other binary operators and therefore will have higher precedence.

6 Optional Reading: Decidability of Ambiguous Grammars

Given a context-free language L, there is no guarantee that an unambiguous grammar such that $L(G) = L$ exists. This was first proven in 1961 by Rohit Parikh. From Wikipedia: An example of an inherently ambiguous language is the union of $\{a^n b^m c^m d^n : n, m > 0\}$ with $\{a^n b^n c^m d^m : n, m > 0\}$. This set is context-free, since the union of two context-free languages is always context-free. But Hopcroft and Ulman in 1979 give a proof that there is no way to unambiguously parse strings in the (non-context-free) common subset $\{a^n b^n c^n d^n : n > 0\}$

In fact, it is not possible to algorithmically recognize whether or not a grammar is ambiguous. Most textbooks (see for example Hopcroft, John; Motwani, Rajeev; Ullman, Jeffrey (2001) Introduction to automata theory, languages, and computation Theorem 9.20, pp. 405-406) reduce to the undecidability of Post's Correspondence Problem. Original proofs are due to Cantor (1962), Floyd (1962), and Chomsky and Schützenberger (1963).

It is possible to algorithmically confirm whether two different DFAs represent the same regular language. The same cannot be said for CFGs, i.e., given 2 CFGs, G_1 and G_2, determining whether or not $L(G_1) = L(G_2)$ is undecidable. Still undecidable is the easier problem of determining whether or not $L(G_1) \cap L(G_2) = \emptyset$.

7 Parsing Algorithms

In Section 4, we discussed that we are not just interested in recognition, i.e., whether a word is in a context-free language. We are interested in obtaining the derivation, since the derivation uniquely specifies the parse tree that represents the program's structure (indeed, ultimately, we're more interested in this parse tree than the derivation per se). The problem of finding the derivation is called `parsing`.

Formally, context-free languages can be recognized by a model of computation called Pushdown Automata. A Pushdown Automaton (PDA) is a Finite Automaton with the addition of a stack, and stack actions on transitions: Each transition may push a symbol to the stack, pop a symbol, and/or require that a given symbol be on the top of the stack. Like Finite Automata, there are Deterministic PDAs and Nondeterministic PDAs. But, while the number of states is finite like Finite Automata, the stack is potentially infinite, and because of this infinite component, there is no conversion of NPDAs to DPDAs equivalent to the conversion of NFAs to DFAs. Without this conversion, it's impractical to actually use PDAs to recognize context-free languages. Instead, we often use specialized parsing algorithms which are less powerful than PDAs, parsing only a subset of CFLs, but which are much more practical.

The goal of these algorithms is to obtain derivations, i.e., given a CFG, $G = (N, T, P, S)$ and a terminal string $w \in T^*$, find a derivation for w or prove that $w \notin L(G)$ (a parser error). Recall our definition of a derivation for w; it is a sequence $\alpha_0 \alpha_1 \cdots \alpha_n$ such $\alpha_0 = S$ (start symbol) and $\alpha_n = w$ (string of terminals) and $\alpha_i \Rightarrow \alpha_{i+1}$ for all $0 \leq i < n$. We have been writing these derivations as $S \Rightarrow \alpha_1 \Rightarrow \cdots \Rightarrow w$.

The next two modules of the course describe two different approaches to writing parsing algorithms.