Towards High-Level Languages: Formal Language Theory
High-Level Languages

• Programs are represented using low-level machine language, which is hard for humans to read and write.

• Assembly language is more convenient, but still low-level, and can be difficult to write and understand.

• Humans sought to develop higher-level languages that make it easier to express what we want our programs to do.

• Common goals of early high level languages were:
  • To be able to write mathematical formulas more naturally. (FORTRAN)
  • To be able to write programs using natural languages like English. (COBOL)
### FORTRAN Example


Given an $N \times N$ square matrix $A$, to find those off-diagonal elements which are symmetric and to write them on binary tape.

<table>
<thead>
<tr>
<th>FORTRAN STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>REWIND 3</td>
</tr>
<tr>
<td>DO 3 I = 1,N</td>
</tr>
<tr>
<td>DO 3 J = 1,N</td>
</tr>
<tr>
<td>IF($A(I,J)-A(J,I)) \neq 0$ 3,20,3</td>
</tr>
<tr>
<td>CONTINUE</td>
</tr>
<tr>
<td>END FILE 3</td>
</tr>
<tr>
<td>MORE PROGRAM</td>
</tr>
<tr>
<td>IF(I-J) \neq 0 20,21</td>
</tr>
<tr>
<td>WRITE TAPE 3,I,J,A(I,J) 21</td>
</tr>
<tr>
<td>GO TO 3</td>
</tr>
</tbody>
</table>
COBOL Example

Here is the complete COBOL program to calculate the answer “2” plus “3.”

```
IDENTIFICATION DIVISION.

PROGRAM-ID. SAMPLE.

ENVIRONMENT DIVISION.

CONFIGURATION SECTION.

SOURCE-COMPUTER. GE-635.

OBJECT-COMPUTER. GE-635.

DATA DIVISION.

WORKING-STORAGE SECTION.

77. ANSWER PIC 9.

CONSTANT SECTION.

77. FIRST-VALUE PIC 9. VALUE IS 2.

77. SECOND-VALUE PIC 9. VALUE IS 3.

PROCEDURE DIVISION.

CALCULATION. COMPUTE ANSWER = FIRST-VALUE +

SECOND-VALUE. DISPLAY ANSWER. STOP RUN.

END PROGRAM.
```
The Grammar of MIPS Assembly

- The "grammar" of MIPS assembly is simple to process line-by-line.
  - Each line starts with zero or more label definitions.
    ➔ Read until a non-label token is found.
  - Then there is optionally an instruction.
    ➔ Match against one of six possible syntax patterns for instructions.
  - Then there is optionally a comment.
    ➔ Check for a semicolon, if found, skip until the end of the line.
  - The only difficult part is matching label uses with their definitions.
    ➔ Solved by doing two passes and building a symbol table.

- Math and natural language are both significantly more complex in structure than assembly, and thus, so are high-level languages.
The Grammar of High-Level Languages

if (say_hello) {
    printf("Hello world!\n");
} else {
    if (a+b*(c+d) == 0) {
        printf("Yay!\n");
    } else {
        printf("I'm sad!!! The number was %d\n", a+b*(c+d));
    }
}

Need to match braces and parentheses (possibly across different lines!), deal with nesting, order of arithmetic operations, distinguish between "if(condition)" and "procedure(arg1, arg2, ...)", and more!
Formal Language Theory

• It is a mathematical approach to describing and studying languages.
• A lot of early development was by linguists who were looking to formalize the structure of *natural languages*.
• Computer scientists found the same ideas useful for formalizing *programming languages*.
• From the early days, there was an interest in finding connections between *models of grammar* and *models of computation*.
  • A formal grammar specifies rules that can be applied to "generate" sentences.
  • The idea was to find models of computation that have the same "generative power" as a certain kind of formal grammar.
"Thus, remarkably, the same important ideas emerged independently for the automatic translation of both natural and artificial languages:

• Separating syntax and semantics.

• Using a generative grammar to specify the set of all and only legal sentences (programs).

• Analyzing the syntax of the sentence (program) and then using the analysis to drive the translation (compilation)."

*Formal Languages: Origins and Directions (S. A. Greibach, 1981)*
Basic Definitions: Alphabets

• An **alphabet** is a finite set. Its elements are called **symbols**.

• $\Sigma = \{a, b, c\}$ is an alphabet containing 3 symbols.
  • Up until now, the symbols in our strings have been single characters...

• $\Sigma = \{\text{cat, dog, mouse, iguana}\}$ is an alphabet containing 4 **symbols**.
  • Individual letters like "c" and "a" are not symbols in this alphabet!

• $\Sigma = \{(x, y) : 0 \leq x, y \leq 9, x \in \mathbb{Z}\}$ is an alphabet containing 100 symbols.
  Each symbol is an ordered pair of integers with values from 0 to 9.

• An alphabet typically cannot be infinite. For example, the set of all integers is not an alphabet (in this course).
Basic Definitions: Strings (Words)

• A string over an alphabet $\Sigma$ is a sequence of symbols from $\Sigma$.
  • Strings are frequently called "words" in formal language theory, but we already use this term for machine-architecture words.

• The length of a string $x$ is denoted $|x|$ and is the total number of symbols in the string (including repeats).

• Examples:
  • $x = \text{cabba}$ is a string over $\Sigma = \{a, b, c\}$. We have $|x| = 5$.
  • $x = \text{cat dog cat dog iguana mouse}$ is a string over $\Sigma = \{\text{cat, dog, iguana, mouse}\}$. We have $|x| = 6$.

• A sequence of zero symbols is allowed. This is called the empty string and it is denoted by the Greek letter $\varepsilon$ (epsilon). We have $|\varepsilon| = 0$. 
Aside about String Notation

• There is no single universal notation for "sequences" in mathematics because the cleanest notation varies wildly depending on context.

• For strings, we often just write them out with no spacing.
  • cat is a string over the alphabet \{a, c, t\}. It has length 3.
  • 11110001 is a string over the alphabet \{0,1\}. It has length 8.

• But if the symbols consist of multiple characters, we might put spaces between each symbol of a string for readability.
  • ID REG COMMA REG is a string over \{ID, REG, COMMA\}. It has length 4.

• Pay attention to what the alphabet is.
  • jr $31$ is a string over the ASCII alphabet. It has length 6.
Basic Definitions: Languages

• A *language* over an alphabet $\Sigma$ is a set of strings over $\Sigma$.
• Equivalently, a language over $\Sigma$ is a subset of $\Sigma^*$.
  • The Kleene star of an alphabet is the set of all strings over the alphabet!
• Strings in a formal language don't necessarily have to be "meaningful" the way strings in a natural language are.
  • "The set of grammatically correct English sentences" is a language, assuming you can agree on what "grammatically correct" means.
  • The set \{sfsfdsdf, fghghfgh, eivlyS\} is also a language.
• The "interesting" classes of formal languages are restricted classes that have more structure than just "a set".
Extending Regular Languages with Recursion

• What if we could use "recursion" in regular expressions?
  • For example: Let L be described by the "recursive regular expression" aLb.
• We can think of this as an equation \( L = aLb \). Is there a language \( L \) that makes this equation true?
• Yes, the language \( \{ a^n b^n : n \geq 0 \} \) (where \( n \) is an integer).
  • The notation \( a^n \) means \( aa...a \) where there are \( n \) occurrences of \( a \).
  • So this language contains words with \( n \) \( a \)'s followed by \( n \) \( b \)'s.
• Can we recognize this language with a DFA? Or describe it with a (non-recursive) regular expression?
The Need for Non-Regular Languages

• It is *impossible* to construct a DFA for \{ a^n b^n : n \geq 0 \}.
• Suppose you had such a DFA, and it had \( m \) states total.
• Run the strings \( a, a^2, a^3, \ldots a^{m+1} \) through the DFA and write down the states you get. Call them \( q_1, q_2, \ldots q_{m+1} \).
• There are only \( m \) states, so two of these are equal. Say \( q_i = q_j \) but \( i \neq j \).
• Since \( a^i b^i \) is accepted, following the sequence of transitions on \( b^i \) from \( q_i \) must lead to an accepting state. So this must be true for \( q_j \) too.
• But \( a^i b^i \) is not accepted since \( i \neq j \)! This is a contradiction, so there is no such DFA!
The Need for Non-Regular Languages

• Does it matter that we can't handle this weird \{ a^n b^n : n \geq 0 \} language? What about something more practical?

• Practical languages are *harder*. Consider the language of *sequences of balanced parentheses* (left brackets matching with right brackets).

• We can make the same argument we just did to show there is no DFA for this language using strings like (((((()))))).

• But this language also contains more complex strings like (())(()(()))).

• Both \{ a^n b^n : n \geq 0 \} and the language of sequences of balanced parentheses are examples of *context-free languages*. 
Context-Free Languages and Grammars

- Context-free languages are conceptually like "regular languages with recursion", but they are usually defined in terms of formal grammars.

- A **formal grammar** has four elements:
  - An alphabet called the *terminal alphabet*, which is the alphabet of the language the grammar is describing.
  - A disjoint alphabet called the *nonterminal alphabet*, which can be thought of as a set of "meta-symbols" that appear in the grammar but not the language.
  - A *start symbol* which is one of the nonterminals (meta-symbols).
  - A set of *production rules*, which are "string rewriting rules", i.e., they tell you that it is valid to replace certain strings of symbols with certain other strings.

- Idea: A string is in the language if you can start with the start symbol and repeatedly apply production rules to eventually obtain the string.
Production Rules

• A production rule, in its most general form, looks like $x \rightarrow y$ where $x$ is a non-empty string and $y$ is a string. This says $x$ can be rewritten as $y$.

• In an "unrestricted" formal grammar, all productions are permitted. The only restriction is that the left hand side is not an empty string.

• In a context-free grammar, the left hand side of each production rule must be a single nonterminal symbol.
  • We can only rewrite "meta-symbols", not actual symbols from the language.
  • "Terminal" refers to the fact that terminal symbols can't be rewritten.
  • Also, we can only rewrite one of these symbols at a time. No surrounding context is allowed in context-free production rules.
Example: \{ a^n b^n : n \geq 0 \} 

• Terminal symbols: \{a, b\}
• Nonterminal symbols: \{S\} (only one is needed)
• Start symbol: S
• Production Rules:
  1. \( S \rightarrow aSb \)
  2. \( S \rightarrow \varepsilon \)

• As an example, we can produce the string \text{aaabbb} by applying rule 1 three times, then rule 2.
• \( S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaabbb \)
Example: Balanced Sequences of Parentheses

• That is, strings over the alphabet \{ (, ) \} where every left parenthesis "(" has a matching right parenthesis ")".

• These rules are not sufficient:
  1. \( S \rightarrow (S) \)
  2. \( S \rightarrow \varepsilon \)

• This doesn't include things like (())

• Adding this third rule is enough (but it's not too easy to prove):
  3. \( S \rightarrow SS \)

• These two rules also work: \( S \rightarrow (S)S \) and \( S \rightarrow \varepsilon \).
Example: \(a(a \mid b)^*a(a \mid b)^*\)

- This is a simple example of a language that requires two distinct nonterminal symbols to describe.

- Terminal symbols: \{a, b\}
- Nonterminal symbols: \{S, B\}
- Start symbol: S
- Production rules: \(S \rightarrow aBaB, \ B \rightarrow aB, \ B \rightarrow bB, \ B \rightarrow \varepsilon\)
Notation & Conventions

• Formally, a context-free grammar is a 4-tuple \((N, \Sigma, S, P)\), where \(N\) is the nonterminal alphabet, \(\Sigma\) is the terminal alphabet, \(S\) is the start symbol, and \(P\) is the set of production rules.

• In practice, we rarely write out all these things separately and instead we use the following convention:
  • A grammar is simply written as a list of production rules.
  • The symbol on the left-hand-side of the first rule in the list is assumed to be the start symbol.
  • If a symbol never appears on the left-hand-side of a rule, it's a terminal symbol. All other symbols are nonterminal.
Notation & Conventions

• To shorten the description of grammars, we use notation that *looks similar* to regular expressions but *means something different*.

• We will often use the | symbol to combine multiple production rules with the same left hand side.

• Example: Instead of $S \rightarrow aBaB, B \rightarrow aB, B \rightarrow bB, B \rightarrow \varepsilon$ we could write $S \rightarrow aBaB, B \rightarrow aB | bB | \varepsilon$

• It is **not valid** to put a regular expression on the right hand side of a production rule. You cannot use star, brackets for grouping, etc.

• This notation is just shorthand for "there are multiple production rules that all have the same right hand side".
Notation & Conventions

• To make things easier to read at a glance, we also have conventions for which kinds of letters correspond to which kinds of objects.
• These aren't strict rules and may be broken sometimes.
• Lowercase letters from the start of the English alphabet (a, b, c, ...) are usually terminals, elements of $\Sigma$.
• Lowercase letters from the end of the English alphabet (..., w, x, y, z) are usually strings of terminals, elements of $\Sigma^*$.
• Uppercase English letters (A, B, C, ..., X, Y, Z) are usually nonterminals, elements of $N$.
• Greek letters ($\alpha$, $\beta$, $\gamma$) are usually strings that may mix terminals and nonterminals, that is, elements of $(N \cup \Sigma)^*$.
"A context-free production rule has the general form $A \rightarrow \alpha$."
  
  • It is implicit that $A$ is a nonterminal, and $\alpha$ is a sequence of symbols that could possibly mix terminals and nonterminals.

• Consider this grammar:
  
  $X \rightarrow aXA \mid bXb \mid Y$
  
  $Y \rightarrow a \mid b \mid \epsilon$

  • It is implicit that the start symbol is $X$.
  
  • Since $a$ and $b$ don't appear on the left hand side of any rule, they are terminals. $X$ and $Y$ are nonterminals.
  
  • There are six production rules in this grammar, not two.
Derivations

• A **derivation** is a sequence of production rules that can be applied to the start symbol of a grammar to produce a string.

• We write derivations by starting with the start symbol, and showing how string evolves with each rule application.

• Example: Find a derivation of (()) in $S \rightarrow (S)S \mid \varepsilon$.
  
  
  \[
  S \Rightarrow (S)S \Rightarrow ((S)S)S \Rightarrow ((S)S)S \Rightarrow ((S)(S))S \Rightarrow ((S)(S)) \Rightarrow (())S \Rightarrow (())
  \]

• In the above example, we always rewrote the *leftmost* nonterminal symbol. Here is another valid derivation:

  
  \[
  S \Rightarrow (S)S \Rightarrow (S) \Rightarrow ((S)S) \Rightarrow ((S)(S)) \Rightarrow ((S)(S)) \Rightarrow (())S \Rightarrow (())
  \]
The Language of a Grammar

• A derivation in a context-free grammar ends once all nonterminals are replaced with terminals.
  • Terminals cannot be replaced or changed, since only nonterminals can appear on the left hand side of a production rule.

• The language generated by a context-free grammar, or just the language of the grammar for short, is the set of all strings of terminals that can be produced as the final result of a derivation.

• A formal language is a context-free language if it is the language of some context-free grammar.