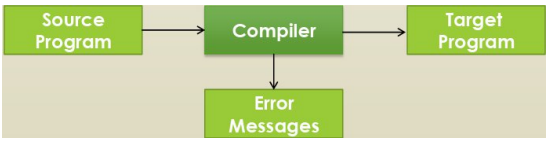
Lecture 1: Introduction.

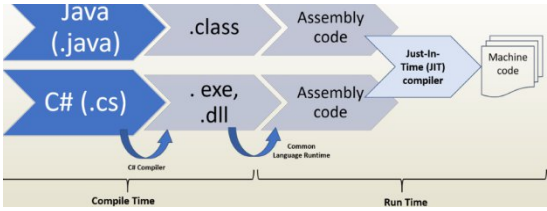
Compiler.

Compiler: a translator that converts high-level languages to machine language.



- Translates source code to target language (mostly assembly).
  - High-level is written by devs.
  - Machine language can be understood by the processor.
- Assists the devs by finding errors in the program during the compile time.
  - Prevents the user from encountering them during runtime.

- Most compilers covert high-level language to an intermediate language (bytecode) and stores it in different ways.
  - Compiler takes source program & produces object program (e.g. .exe).
  - The assembler takes the object program & produces the target program.



Compile-time.

- Compile-time: the time it takes to compile the source program.
  - The compiler checks the code type and looks for syntax & semantic errors.
  - Input: source code, dependent files, interfaces, required libraries.
  - Output: compiled code (assembly code || relocatable object code) || compile time error messages.
  - Compilation errors: syntax or semantic errors that occur during compile time.
    - Syntax error: errors caused by wrong syntax.
    - semantic error: errors related to variables, functions, type declaration, and type checking.

Runtime.

- Runtime: the time it takes to load & execute the object program.
- Runtime is the program's life cycle during execution.
- Some runtime errors:
  - Zero division.
  - Dereferencing a null pointer: attempting to access memory containing nothing (null).
  - Running out of memory.

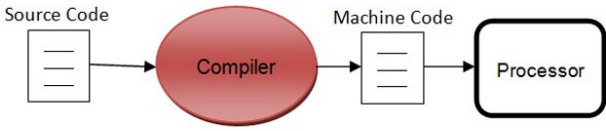
```
x = a-a;
y = 100/x;          // division by 0

Integer n[ ] = new Integer[7];
n[8] = 16;          // invalid subscript
```

Compile time	Runtime
Time period for translation of a source code like Java to intermediate code like .class	Time period between start and end of running intermediate code at runtime environment
This is to check the syntax and semantics of the code	This is to run the code
Errors get detected by compiler without execution of the program	Only can detect after execution of the program
Fixing an error at this stage is possible	Fixing an error requires going back to code

Compilers & Interpreters.

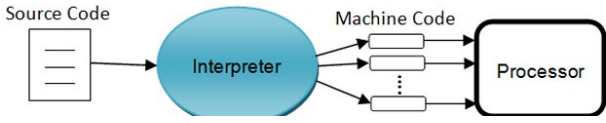
Compilers:

- Scan the entire program & translates all of it into machine code.
  - Java translates Java source code into java bytecode.
    - Java bytecode is an abstract form of assembly.
    - Some implementations of java work as interpreters.
  - Just in time compiling (JIT): translates bytecode into local machine code & runs the code directly.



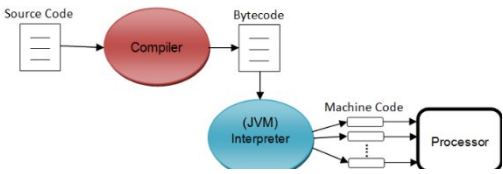
Interpreters:

- Scans the source program instruction by instruction.
- It translates each instruction into machine code and executes it.
- Python & ruby use interpreters.



Interpreter compiler hybrid:

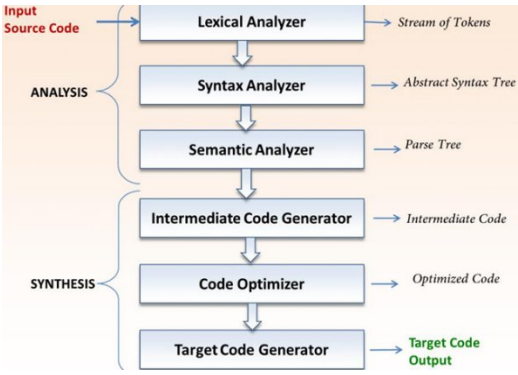
- Compilers & interpreters can be combined together.
- The compiler produces intermediate-level code.
- The interpreter interprets the generated code into machine code.



Interpreter	Compiler
Translates program one statement at a time.	Scans the entire program and translates it as a whole into machine code.
It takes less amount of time to analyze the source code but the overall execution time is slower.	It takes large amount of time to analyze the source code but the overall execution time is comparatively faster.
doesn't save the translated code in a file	saves the translated code in a file (.object file)
Continues translating the program until the first error is met, in which case it stops. Hence debugging is easy.	It generates the error message only after scanning the whole program. Hence debugging is comparatively hard.
Programming language like Python, JavaScript, Perl use interpreters.	Programming language like C#, C++ use compilers.

Compiler phases:

- Compilation process is a sequence of phases.
- The first phase input is the source code.
- Each phase after that takes its input from the output of the phase before it.



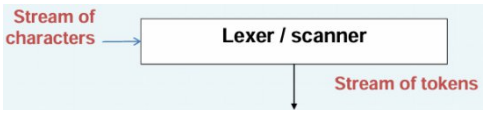
High level languages.

- Using high level languages impacts how fast programs are developed.
- Reasons to use high level languages:
  - Their notation is closer to the way humans think.
  - Compiler can spot some obvious mistakes.
  - Programs tend to be shorter.
  - The same program can be compiled in many different machine languages (can be run on different machines).

Lecture 2: Lexical Analysis.

Lexical Analysis.

- Lexical analyzer (Scanner): reads a stream of characters & puts them into tokens.
  - The first compiler phase.
  - Tokens are the input of the next compiler phase.



- Tokens: some meaningful unit.
  - A token is a pair of (pattern, attributes).
- Token parts:
  - A class indicating the token kind.
  - A value indicating the class member.
    - Some token classes might not have a value part.

```
[num: 4]
[op: '*']
```

- Tokens can be:
  - Keywords (e.g. while, void, etc.).
  - Identifiers: declared by the programmer (variables).
  - Operators (e.g. +, /, =, etc.).
  - Numeric constants: any constant number.
  - Character constants: any constant character or string.
  - Special characters (e.g. ;, :,), (, etc.).
  - Comments: ignored by the next phases.
    - Must be identified by the scanner but it isn't included in the output.

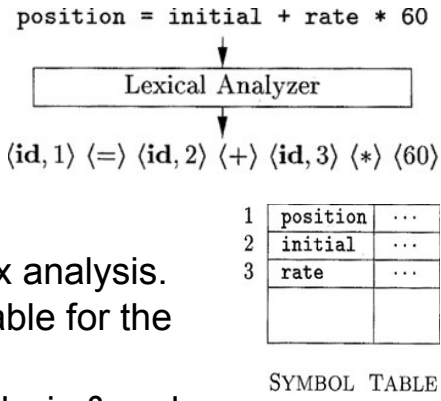
- Lexeme (symbol): a series of letters separated from the rest of the program (the character sequence forming a token).
  - Separation follows some rule (e.g. space, semi-column, comma, etc.).

```
int maximum(int x, int y) {
```

Lexeme
int
maximum
(
int
x

- Pattern: a rule specifying a set of strings.
  - Usually it's a regular expression.
  - E.g. "an identifier is a string that starts with a letter".

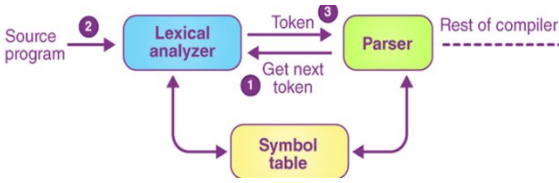
Lexical analyzer steps:



- Read and analyze the program text.
- Divide the program into lexemes.
- For each lexeme:
  - Produce a token in the form <token-name, attribute-value>.
  - Token-name: an abstract symbol used during syntax analysis.
  - attribute-value: a pointer to an entry in the symbol table for the token.
    - Info in the symbol table is used in semantic analysis & code generation.
    - Symbol table stores identifiers used in the source program & includes relevant information and attributes.

Lexical analyzer architecture.

- Its main task is to read code characters & produce tokens.
- It scans the entire source code & identifies each token one by one.
- Scanners are usually implemented to produce tokens when requested by a parser.
  - Parser sends “get next token” command.
  - Lexical analyzer scans the input until it finds the next token.
  - It returns it to the parser.



- Lexical analyzer skips whitespaces & comments while creating tokens.
- If there’s an error, it will link the error with the source file & line number.
- Lexical analyzer tasks:
  - Reads input characters from the source program.
  - Helps identify tokens into the symbol table.
  - Removes whitespace & comments.
    - Also skips pre-processor directive (e.g., include, define).
  - Links error messages with the source program.

Lexical errors.

- lexical error: a character sequence which isn't possible to scan into a valid token.
- They aren't very common, but should be managed by the scanner.
- Misspelling identifiers, operators, or keywords are considered errors.
- Generally caused by the appearance of some illegal character.
- Some errors aren't detectable at the lexical level alone.
  - E.g., "fi (num == 3)", it cant tell whether "fi" is an identifier or a misspelling of "if".
- The lexical analyzer can detect chars that're not in the alphabet & strings with no pattern.

- Error recovery techniques:
  - Removing one character from the remaining input.
  - In panic mode, successive characters (e.g. iff) are always ignored until a well formed token is reached.
  - Inserting the missing character into the remaining input.
  - Replace a character with another.
  - Transpose two serial characters.

Regular languages (RL).

- Regular expressions (RE): an algebraic notation that describes sets of strings.
- Token specifications are written using regular expressions.
  - The generated lexers are in a class of simple programs called finite automata (FA).
- RLs are defined equivalently by REs & finite-state automata.
- Notations:
  - Σ à finite set of letters.
  - Word à sequence of letters.
  - Language à set of words.
  - – à a range (e.g., 0 – 9).
  - * à star closure (allowed to iterate).

Common format for reg-exps

Basic Patterns	Matching
x	The character x
.	Any character, usually except a new line
[xyz]	Any of the characters x,y,z
^x	Any character except x
Repetition Operators	
R?	An R or nothing (=optionally an R)
R*	Zero or more occurrences of R
R+	One or more occurrences of R
Composition Operators	
R1R2	An R1 followed by R2
R1|R2	Either an R1 or R2
Grouping	
(R)	R itself

RE handling examples.

- Keywords: described by a regular expression that looks like the keyword.
  - Keyword ‘if’ = RE ‘if’ (made of the concatenation of RE ‘i’ & RE ‘f’).

- Variable names: a variable must start with a letter || an underscore and can contain letters, digits, and underscores.
  - RE à [a-z | A-Z | _] [a-z | A-Z | 0-9 | _]*.

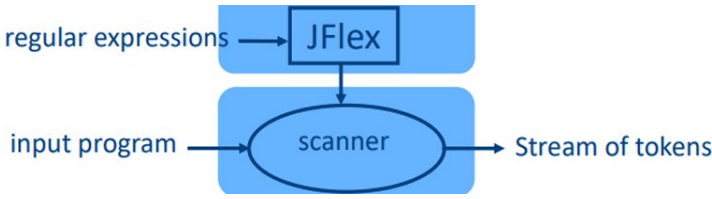
- Integers: a sequence of digits with an optional sign.
  - RE à[+ | -]? [0-9]+.

- Floats: a sequence of digits with an optional sign & decimal point.
  - RE à[+ | -]? [0-9]+ [[.] [0-9]+]?.

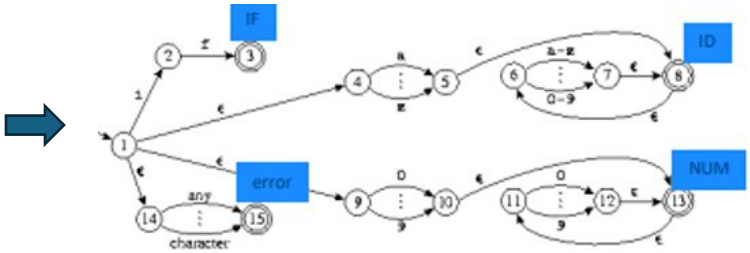
- String constants: can include letters or digits & has some special “escape” sequences (e.g., \n).
  - RE à “([a-z | A-Z | 0-9] | \ [a-z | A-Z])*”.

Program generating tools.

- Input:
  - regular expressions & actions.
    - Action = java code.
- Output:
  - a scanner program that:
    - produces a stream of tokens.
    - Invokes actions when a pattern is matched.



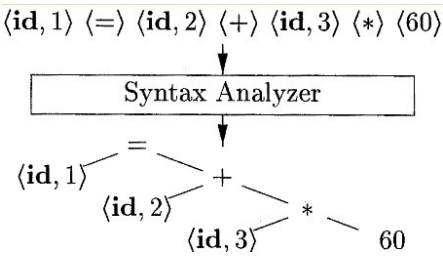

```
if                  { return IF; }
[a-z][a-z0-9]*      { return ID; }
[0-9]+              { return NUM; }
```



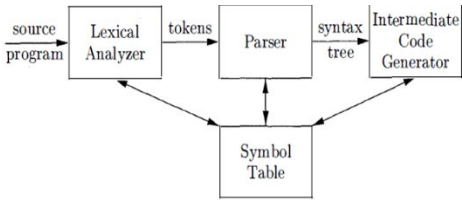
Lecture 3: Syntax analysis.

Syntax analysis (Parsing).

- A parser takes a stream of tokens & determines if the syntax (structure) is correct according to the context-free grammar (CFG) of the language.
- It generates a parse tree || an abstract syntax tree (AST).
  - Describes the program's syntax structure.
  - Interior node = operation.
  - Children = arguments of an operation.



- If a tree cant be build, the syntax isn't correct.
- Units (tokens): identifiers handled by a translator during syntax analysis.
- Not all strings of tokens are programs.
- Parser must distinguish between valid & invalid strings of tokens.
  - CFG: a language used to describe valid strings of tokens.



- Parser functions:
  - Verifying token generated structure using CFG.
  - Constructing a parse tree.
  - Reporting syntax errors.
  - Performing error recovery using recovery techniques.

- Error types (compile time):
  - Lexical: misspelling identifiers, keywords, or operators.
  - Syntactical: missing semicolon, or unbalanced structure.
  - Semantical: incompatible value assignment, or type mismatch.
  - Logical: code not reachable, or infinite loops (cant be detected by compilers).

- Errors undetectable by the parser (handled by the semantic analysis):
  - Variable re-declaration.
  - Variable initialization before use.
  - Data type mismatch.

CFG, RE, and NFA.

- REs are a nice formalism, but they cant describe all languages.
  - E.g., $\{0^n1^n \mid n>= 0\}$ cant be described using NFAs || REs.
- Regular language (RL): a language described by a regular expression.
- CFGs provide a more powerful mechanism for language specification.
  - It can describe features that have a recursive structure.

- RE:
  - Used to describe tokens & check whether the input is valid using a transition diagram.
    - Transition diagram has a set of states & edges.
  - Doesn't have a start symbol.
  - Useful for describing the structure of lexical constructs (e.g. identifiers, constants, keywords, etc.).

- CFG:
  - Used to check whether the input is valid using derivation.
  - Has a set of production rules.
  - Has a start symbol.
  - Useful for describing nested structures (e.g. balanced parentheses, matching begin-end's, etc.).

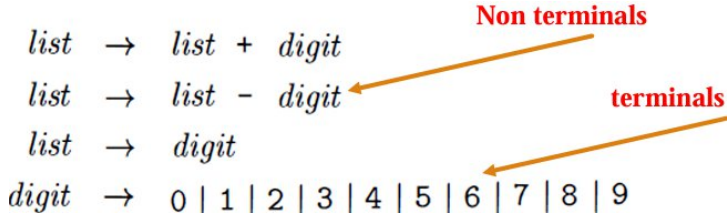
Context free grammar.

- CFG Consists of 4-tuples:
  - V: finite set of variables (Non-terminal symbols).
  - $\sum$ (T): set of terminal symbols (lowercase letters) of the language's alphabet.
    - $V \cap \sum = \text{NULL}$.
  - S: the start symbol (non-terminal).
  - P: finite set of production rules.
    - Form: α è β.
    - α & $\beta \in (V \cup \sum)$.
    - At least one symbol in $\alpha \in V$.

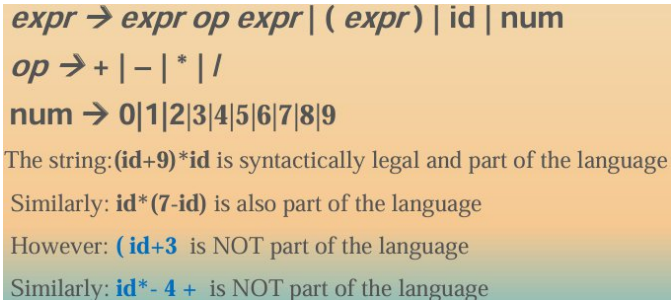
The grammar ({A}, {a, b, c}, A,P), P : A → aA, A → abc.

The grammar({S},{a, b},S,P), P: S → aSa, S → bSb, S→ε

The grammar({S,F},{0, 1},S,P) ,P: S→00S | 11F, F → 00F|ε

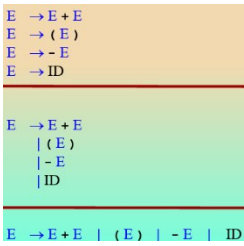


- Language (L) is a context free language (CFL), if there's a CFG that generates it.
  - L = L(G).
- A grammar derives strings by:
  - Beginning with the start symbol.
  - Repeatedly replacing non-terminals by a production rule.
- Sentential forms: a sequence of terminal and non-terminal in a derivation (e.g. à [id * E]).
- Two CFGs are equivalent, if they generate the same language.
  - $G_1 \approx G_2$, if $L(G_1) = L(G_2)$.



Rules & notations.

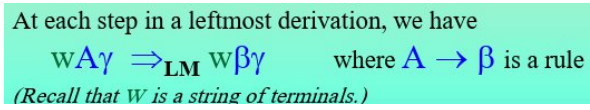
- Rules:
  - A production rule that has more than one right side result can be written separately, or written together but separated by '|'.



- Notation conventions:
  - Terminals are lowercase letters.
  - Non-terminals are uppercase letters || begin with an uppercase.
  - Strings of symbols: a sequence of zero or more terminals & non-terminals.
  - Strings of terminals: sequence of terminals together (includes null).

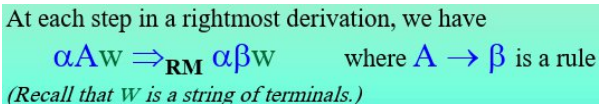
Leftmost & Rightmost derivatives.

- Leftmost:
  - While deriving, always expand the leftmost non-terminal symbol.
  - Each sentential form in leftmost derivation is called left-sentential form.

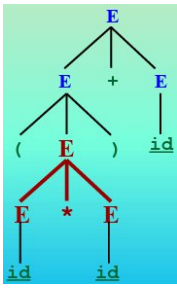


- Rightmost:
  - In a derivation, always expand the rightmost non-terminal symbol.

- Each sentential form in rightmost derivation is called right-sentential form.



Parse trees.



- Derivations (rightmost || leftmost) have parse trees associated with them.
- Sometimes a rightmost tree looks the same as the leftmost tree.
- Parse tree only remembers which rule you are replacing a non-terminal with.
- Every parse tree corresponds to a single, unique derivation (left || right).
- One input string may have multiple parse trees (ambiguity).

Ambiguous grammar.

- Ambiguous grammar: grammar whose sentences have more than one parse tree.
- Programming languages grammar may be ambiguous.
  - Needs to be modified before parsing.
- Grammar may be left recursive (needs to be modified).

RE to CFG conversion.

- First build the RE's NFA.
- For every state in the NFA:
  - add a non-terminal in the grammar.
- For every edge labeled (c) from node A to node B:
  - add rule A à cB.
- For every edge labeled (ε), from node A to node B:
  - add rule A à B.
- for every final state B:
  - add rule B à ε.

Parsing types.

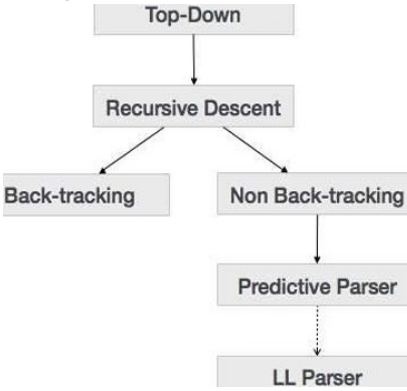
- Top-down parsing:

- Starts with the start symbol & tries to transform it into the input string.
- Tree is constructed:
  - From the top.
  - From left to right.
- Terminals are added in order of appearance.

- Bottom-up parsing:
  - Starts with the input string & tries to rewrite it into the start symbol.

Top-down parsing.

- Recursive decent parsing: a common form of top-down parsing.
- Uses recursive procedures to process the input.
- Suffers from backtracking.
- Backtracking: if one derivation fails, the syntax analyzer restarts the process using different rules of the same production.
  - May process the input string more than once.
- Predictive parsing: a form of recursive decent that doesn't require backtracking.



- Steps:
  - Find a leftmost derivation.
  - Start building the tree from the root & work down.
  - Choose which rule to use & where to use it.

- Top-down options:
  - Option 1:
    - Use backtracking if you make a bad decision.

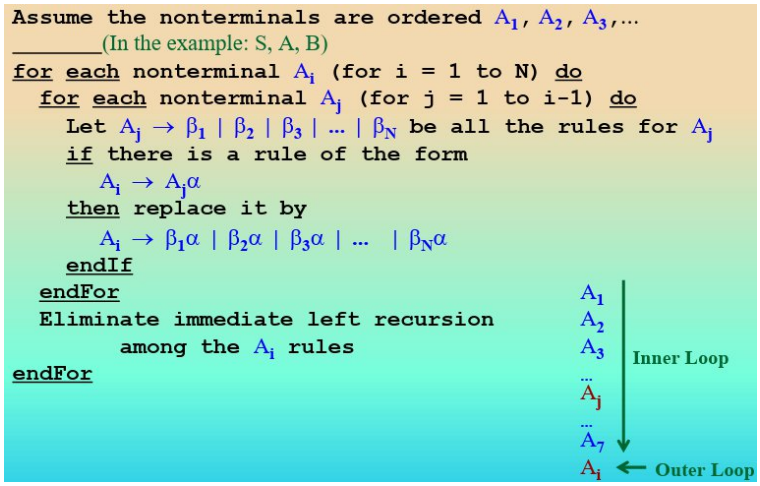
  - Option 2:
    - Always make the right decision.
    - Never backtrack (predictive parsing).
    - Only possible for some grammars (LL grammars).

        - May be able to fix some grammar problems (eliminate left recursion, left factor the rules).
- Examples: page 36 – 57.

- Recursive decent limitations:
    - They only work on grammars with certain properties.
        - Doesn't work on grammars containing left recursion || left factoring (leads to an infinite loop).

Left recursive grammar.

- Left recursion: a production rule where the leftmost variable in its RHS is the same as the LHS.
    - E.g., E à E + T.
- Left recursive grammar: a grammar with a left recursion rule.
- Left recursion elimination (Convert left recursion to right recursion):
    - If we have the rule A àAα | β (left recursive), where β doesn't begin with an A.
    - Replace the pair with:
        - A à βA/.
        - A/ à α A/ | ε.



Lecture 4: Syntax Analysis (Cont'd).

Predictive parsing.

- Never backtracks.
- Requirement:
  - May peek ahead at the next symbol (token).
  - For every rule (A ®a1| a2 | a3| ... | a_n):
    - We must be able to choose the correct alternative by looking only at the next symbol.

- LL(1) grammars:
  - Can do predictive parsing.
  - LL (1) grammars: A subset of CFGs that are easy to parse with simple algorithms.
    - A grammar is LL(1) if it can be parsed by considering only one non-terminal & the next token in the input steam.
  - To make sure a grammar is LL(1):
    - Remove any ambiguity.
    - Eliminate any left recursion.
    - Eliminate any common left prefixes.

- LL(k) grammars:
  - Can do predictive parsing.
  - Can select the right rule by looking at the next k input symbols.
- LL(k) language: A language that can be described with a LL(k) grammar.

- Techniques to modify grammar:
  - Left factoring.
  - Removal of left recursion.

- If-then problem:
  - When a if-then-else rule is inside a if-then rule, the 'else' part can belong to the inner or outer if.
  - In Programming languages, the 'else' part belongs to the inner if (grammar cant reflect that).

Eliminating Left Factorization (common left prefixes).

- Common left prefixes: a simple problem that appears when multiple rules have a common left prefix.
- Solution:
  - Replace them with one rule that contains the prefixes and another containing the variants (RHS).
- Can be used to solve the if-then problem.

Before: $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3 \mid \ldots \mid \delta_1 \mid$

After: $A \rightarrow \alpha C \mid \delta_1 \mid \delta_2 \mid \delta_3 \mid \ldots$

$C \rightarrow \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots$

Table driven predictive parsing algorithm.

- Assumes the grammar is LL(1).
- Doesn't use backtracking.
- Always knows which RHS to choose.
  - No left recursion.
  - Grammar is left factored.
- Steps:
  - Construct a table from the grammar (Link).
  - Use table to parse the input (Link).
  - Construct the abstract parse tree (AST).