What is a program?

- A program is a sequence of characters
- But what is a legal program?
  - Could have a list of all legal programs
  - Better to have rules
- Similar to spoken and written languages
  - Syntax and grammar rules – punctuation, words in the dictionary, rules for sentence construction

Lexical analysis (scanning)

- Group characters into token types, e.g., arithmetic operations, keywords, variable names
- These are the "words" of the programming language
- Done by pattern matching (regular expressions)
- Well understood how to construct scanners
  - Tools like lex and flex to automate scanner generation
Syntax analysis (parsing)

- Scanning turns stream of characters into a stream of tokens, but how to tell if it makes sense?
- Needs to have some structure
  - Like written documents have structure of sentences, punctuation, and paragraphs
  - Called the grammar of the language
- Result of parsing is a parse tree
  - Data structure that reflects the grammatical organization

Formal Definition of Grammar

A Context Free Grammar consists of

- A set of terminal symbols
- A set of non-terminal symbols (grammar variables)
- A set of production rules
- A non-terminal designated as the start symbol

A production rule is a mapping from a non-terminal to a string of terminals and non-terminals. Empty string is usually specified as $\varepsilon$. 
Grammars

- A CFG specification precisely defines a language
- But it is not always the easiest thing to read
- Informal descriptions, user guides, examples are often better ways to understand the syntax of a language
- But the CFG is not subject to interpretation
  - Used by compiler writers
  - Allows formal analysis of the language

Backus Naur Form (BNF)

Convention for writing down a grammar

- **Distinguish terminals from non-terminals**
  - Use angle brackets for non-terminals, or quotes for terminals
  - Sometimes use capitals for non-terminals
- **Specify production rules with** $::=$ **or arrow separating left hand side (non-terminal) and right hand side (replacement string of terminals and non-terminals)**
- **Use alternation symbol | to combine rules that have same left side**
- **Some other extensions (EBNF) for optional parts**
Example of a CFG

Language with 3 variables, arithmetic, assignment

Terms: \{ A, B, C, +, -, =, ; \}
Non-terminals: \{ <stmt>, <var>, <expr> \}
Start symbol: <stmt>

Production rules:
\[ <stmt> ::= <var> = <expr> ; \]
\[ <var> ::= A | B | C \]
\[ <expr> ::= <var> + <var> | <var> - <var> | <var> \]

How does a grammar work?

- Think of it as a program generator
  - Begin with start symbol
  - Pick a rule and replace symbol by right side
  - For each non-terminal in result, repeat
  - When only terminals left, we have a legal program
  - The result is a production and the sequence of substitutions is called a derivation
- A program is a legal program if there is a derivation that yields the program, for example:
\[ <stmt> \rightarrow <var>=<expr>; \rightarrow A=<expr> ; \]
\[ \rightarrow A=<var>+<var>; \rightarrow A=B+<var>; \rightarrow A=B+C; \]
More examples

Sequence of a's followed by equal number of b's

\[ T = \{ a, b \} \quad N = \{ S \} \quad \text{start} = S \]

Production rules:

\[ S \rightarrow aSb \mid ab \]

Every a is followed by a b

\[ T = \{ a, b \} \quad N = \{ S, X, Y \} \quad \text{start} = S \]

Production rules:

\[ S \rightarrow aX \mid bS \mid \varepsilon \]
\[ X \rightarrow bY \mid bS \]
\[ Y \rightarrow aX \mid bY \]

Simplify to:

\[ S \rightarrow abS \mid bS \mid \varepsilon \]

Another representation of derivation

- Draw the derivation as a parse tree
  - Internal nodes of tree are non-terminals
  - Rule choice determines children of a node
  - Leaves are terminals
  - Read leaves from left to right to get the production

\[
<stmt> \\
<var> = \\
<expr> \\
<var> + <var> \\
A \quad B \quad + \quad C \quad ;
\]
Uniqueness of grammars

- Many grammars may generate the same language
  \[
  \text{<stmt>} ::= \text{<var>} = \text{<expr>} ; \\
  \text{<var>} ::= A \mid B \mid C \\
  \text{<sum>} ::= \text{<expr>} + \text{<var>} \\
  \text{<diff>} ::= \text{<expr>} - \text{<var>} \\
  \text{<expr>} ::= \text{<sum>} \mid \text{<diff>} \mid \text{<var>}
  \]

- Produces the same language, but parse trees for a given string would be different

```
program ::= decl_and_process_list
decl_and_process_list ::= declarations \mid process
process ::= decl_and_process_list declarations \mid decl_and_process_list process
statement_list ::= statement \mid declarations \mid statement_list statement \mid statement_list declarations
body ::= LBRACE statement_list RBRACE
statement ::= expr SEMI \mid SEMI \mid IF LPAREN expr RPAREN statement \mid IF LPAREN expr RPAREN statement ELSE statement \mid SWITCH LPAREN expr RPAREN LBRACE case_list RBRACE \mid FOR LPAREN expr SEMI expr SEMI expr RPAREN body \mid WHILE LPAREN expr RPAREN body \mid CONTINUE SEMI \mid BREAK SEMI \mid CONTINUE SEMI \mid EXIT SEMI \mid EXIT LPAREN RPAREN \mid EXIT expr \mid SESSION\_RETURN expr \mid SESSION\_RETURN SEMI \mid TRY cstatement catch_list FINALLY cstatement \mid TRY cstatement catch_list \mid THROW ID \mid THROW ID LPAREN RPAREN \mid THROW ID LPAREN expr \mid THROW ID LPAREN expr RPAREN SEMI
```
Grammar Ambiguity

Give me ambiguity or give me something else...

- Do we always get the same derivation or same tree?
- Derivations could certainly be different depending on the order of applying rules to replace non-terminals
  - But we can talk about left-most and right-most derivations (e.g., replace left-most non-terminals first)
- Trees don't have the problem of order
- If there are two distinct trees for a string produced by a grammar, then we say the grammar is ambiguous

Ambiguity Example

\[
\begin{align*}
\langle \text{assign} \rangle & \rightarrow \langle \text{id} \rangle = \langle \text{expr} \rangle \\
\langle \text{id} \rangle & \rightarrow A \mid B \mid C \\
\langle \text{expr} \rangle & \rightarrow \langle \text{expr} \rangle + \langle \text{expr} \rangle \\
& \mid \langle \text{expr} \rangle \times \langle \text{expr} \rangle \\
& \mid ( \langle \text{expr} \rangle ) \\
& \mid \langle \text{id} \rangle
\end{align*}
\]

Consider the string \( A = B + C \times A \)
Ambiguity Example

Two different trees for the same string, so the grammar is ambiguous.

Ambiguity Concerns

- Why do we care if the grammar is ambiguous?
  - The parse tree represents a structure for the string
  - In the example, the string has an expression to evaluate
  - The parse tree structure is what we use to know how to evaluate this expression
  - It would be reasonable to recurse through the tree to evaluate, evaluating subtrees to get the values for the parent
- The difference here would result in different precedence for multiplication versus addition, and thus different evaluations
Dealing with ambiguity

• Could require the user to always include parentheses
  - It would make the language less readable
  - It would also make the language less writable
  - It would make the language less reliable
  - If user forgot, one implementation would work one way, another implementation might work another way

• Languages should not have surprises

Dealing with ambiguity

• A better way - rewrite the grammar to not be ambiguous
  <assign> ::= <id> = <expr>
  <id> ::= A | B | C
  <expr> ::= <expr> + <term> | <term>
  <term> ::= <term> * <factor> | <factor>
  <factor> ::= ( <expr> ) | <id>
Ambiguity Resolution

This is the only tree for the string, and it properly reflects the precedence of the operators.

Fixing Ambiguity

- Often caused by multiple appearances of symbols in rules that allow too many distinct choices
- Technique is to add rules to separate the symbols so choices are unique
- Grammar can also be designed to reflect associativity as well as precedence of operators
- Ambiguity in the grammar can be tolerated if it does not lead to ambiguity in the interpretation of the language
- May also be fixed by ad hoc rules in the parsing process
Another Ambiguity Example

Fragment of a grammar for conditional statements in C

```
<if-stmt> → if <expr> <stmt>
         | if <expr> <stmt> else <stmt>
<stmt> → <if-stmt>  | S1  | S2
```

Consider the statement
if <expr> if <expr> S1 else S2

Dangling else Example

```
if <expr>  if <expr>  S1  else  S2
if <expr>  S1
else  S2
```
How to fix dangling else?

• Add the keyword 'endif' to constrain the clause
  • But this would change the language and may not be acceptable
• Don't allow an if without an else
  • This really changes the language
  • ML does this, but if-else is an expression, not a statement there
• Use add hoc rules in the parsing and document the behavior (e.g., else goes with nearest if)
• Fix the grammar…

Unambiguous if-else

<matched> → if <expr> else <matched> | S1 | S2
<unmatched> → if <expr> <stmt>
  | if <expr> <matched> else <unmatched>
<stmt> → <matched> | <unmatched>
More on ambiguity

- It may not be possible to rewrite the grammar to be unambiguous
- If there is no unambiguous grammar for a language, the language itself is said to be inherently ambiguous
  - Proving an arbitrary language is ambiguous is an undecidable problem
  - However, specific languages can sometimes be proved to be inherently ambiguous

Parsing

- It's one thing to generate a string from a grammar, but how do you tell if a given string is generated?
  - Could keep generating strings until we get our string
  - Time consuming and there is a problem that there may be an infinite number of strings generated
- The parsing problem is well understood
  - Various parsing techniques (depending on characteristics of the grammar)
  - Parser generators (yacc/bison)
Top Down Parsing

- Predictive parsing
- Builds parse tree from root down
  - Looks at next token to decide which rule to use
  - Grammar must allow this decision to be made unambiguously
- Can hand code recursive descent parsers

Top Down Parsing Example

- Grammar for comma separated value list ending with ;
  
  \[
  \text{csv} \rightarrow \text{val} \; \text{csv-tail} \\
  \text{csv-tail} \rightarrow , \; \text{val} \; \text{csv-tail} \mid ;
  \]
- Is the string \text{A,B,C;} generated by this grammar?
Bottom Up Parsing

- Builds parse tree from the leaves up
  - Looks at input and tries to combine leaves into a node via a rule
  - These nodes along with remaining leaves can be combined
  - Eventually we produce the start node
  - Process is one of reduction
  - May require look ahead to make decision
  - May need to process to end of input before nodes can be built
- Yacc generates parsers for LALR grammars

Bottom Up Parsing Example

- Same grammar
  
  \[
  \text{csv} \rightarrow \text{val} \text{ csv-tail} \\
  \text{csv-tail} \rightarrow , \text{val} \text{ csv-tail} | ;
  \]

- Go through input until it can be reduced via a rule
A better grammar

- Same language, different grammar
  
  $$\text{csv} \rightarrow \text{csv-prefix} ;$$
  
  $$\text{csv-prefix} \rightarrow \text{csv-prefix} , \text{val} | \text{val}$$

- Nodes get built as we go along

```
  csv
  \---------->
  csv-prefix
  \---------->
  csv-prefix
  \------>
  csv-prefix
  \------>
  csv-prefix
  \------>
  A   ,   B   ,   C   ;
```