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)
  - Longest substring
  - Token separators, e.g., whitespace which is ignored
- Well understood how to construct scanners
  - Tools like lex, flex and jflex 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
  - E.g., 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

Terminals:  \{ A, B, C, +, -, =, ; \}
Non-terminals:  \{ stmt, var, expr \}
Start symbol: stmt
Production rules:

\[
\begin{align*}
\text{stmt} & \ ::= \ var = \ expr \ ; \\
\text{var} & \ ::= \ A \mid B \mid C \\
\text{expr} & \ ::= \ var + \ var \mid var - var \mid var
\end{align*}
\]

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:

\[
\begin{align*}
\text{stmt} & \rightarrow \ var = \ expr \ ; \\
& \rightarrow \ A = \ expr \ ; \\
& \rightarrow \ A = \ var + \ var \ ; \\
& \rightarrow \ A = B + \ var \ ; \\
& \rightarrow \ A = B + C \\
\end{align*}
\]
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:

Every a is followed by a b
$T = \{ a, b \} \quad N = \{ S, X, Y \} \quad \text{start} = S$

Production rules:

Simplify to:

Basic Example: Simple integer arithmetic expressions

exp $\rightarrow$ exp op exp $|$ ( exp ) $|$ number

op $\rightarrow$ $+$ $|$ $-$ $|$ $*$

In what way does such a CFG differ from a regular expression?

digit = 0|1|...|9

number = digit digit*

Recursion!
Recursion

• Recursion is a natural way to define program structures
  • Expressions are made up of expressions
  • Structure of nested blocks is recursively defined…
  • Data structures are collections of data structures…
  • Multi-dimensional arrays are arrays of arrays…
  • Nested parentheses, braces, brackets…

• Thus, we need CFGs to define programming languages

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

```
A = var B + var C
```
Uniqueness of grammars

- Many grammars may generate the same language

\[
\begin{align*}
stmt & ::= \text{var} = \text{expr} ; \\
\text{var} & ::= A \mid B \mid C \\
\text{sum} & ::= \text{expr} + \text{var} \\
\text{diff} & ::= \text{expr} - \text{var} \\
\text{expr} & ::= \text{sum} \mid \text{diff} \mid \text{var}
\end{align*}
\]

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

Sequence Examples

- **one** or more stmts **separated** by a semicolon

\[
\text{stmt-seq} \rightarrow \text{stmt} ; \text{stmt-seq} \mid \text{stmt}
\]

- **zero** or more stmts **terminated** by a semicolon

\[
\text{stmt-seq} \rightarrow \text{stmt} ; \text{stmt-seq} \mid \varepsilon
\]

- **one** or more stmts **separated** by a semicolon

\[
\text{stmt-seq} \rightarrow \text{stmt-seq} ; \text{stmt} \mid \text{stmt}
\]

- **zero** or more stmts **preceded** by a semicolon

\[
\text{stmt-seq} \rightarrow \text{stmt-seq} ; \text{stmt} \mid \varepsilon
\]
program ::= decl_and_process_list
   decl_and_process_list ::= declarations | process
   | decl_and_process_list declarations | decl_and_process_list process
process ::= SESSION ID body | SESSION ID LPAREN arg_list RPAREN body
statement_list ::= statement | declarations | statement_list statement | statement_list declarations
body ::= LBRACE statement_list RBRACE
statement ::= expr SEMI
   | expr
   | IF LPAREN expr RPAREN statement
   | IF LPAREN expr RPAREN statement ELSE statement
   | SWITCH LPAREN expr RPAREN LBRACE case_list RBRACE
   | FOR LPAREN expr SEMI expr SEMI expr RPAREN statement
   | WHILE LPAREN expr RPAREN expr RPAREN statement
   | CONTINUE SEMI
   | BREAK SEMI
   | EXIT SEMI
   | EXIT LPAREN RPAREN SEMI
   | EXIT expr SEMI
   | SESSION_RETURN expr SEMI
   | TRY cstatement catch_list FINALLY cstatement
   | TRY cstatement catch_list
   | THROW ID SEMI
   | THROW ID LPAREN RPAREN SEMI
   | THROW ID LPAREN RPAREN 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

assign → id = expr
id → A | B | C
expr → expr + expr
    | expr * expr
    | ( expr )
    | id

Consider the string  A = B + C * 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
  
  \[
  \begin{align*}
  \text{assign} & ::= \text{id} = \text{expr} \\
  \text{id} & ::= A \mid B \mid C \\
  \text{expr} & ::= \text{expr} + \text{term} \mid \text{term} \\
  \text{term} & ::= \text{term} \ast \text{factor} \mid \text{factor} \\
  \text{factor} & ::= (\text{expr}) \mid \text{id}
  \end{align*}
  \]

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

\[
\text{if-stmt} \rightarrow \text{if expr stmt} \\
\quad \quad | \quad \text{if expr stmt else stmt} \\
\text{stmt} \rightarrow \text{if-stmt} \quad | \quad \text{S1} \quad | \quad \text{S2}
\]

Consider the statement

\text{if expr if expr S1 else S2}
Dangling else Example

```
if expr
   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

\[ \text{matched} \rightarrow \text{if} \ \text{expr} \ \text{matched} \ \text{else} \ \text{matched} \mid S1 \mid S2 \]
\[ \text{unmatched} \rightarrow \text{if} \ \text{expr} \ \text{stmt} \]
\[ \quad \mid \text{if} \ \text{expr} \ \text{matched} \ \text{else} \ \text{unmatched} \]
\[ \text{stmt} \rightarrow \text{matched} \mid \text{unmatched} \]

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/CUP)

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

```
  csv
     \--- csv-tail
         \--- csv-tail
             \--- csv-tail

  A , B , C ;
```

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

  A , B , C ;
```