Languages

What do we mean by a programming language?
What are the legal programs in the language?
What is the meaning of a legal program?
Syntax refers to the rules that describe what is legal
Semantics refers to the meaning
Definitions used in defining a Language

An alphabet is a finite set of symbols

A terminal symbol is a symbol that is a member of the alphabet set

A nonterminal symbol is any other symbol (usually used in defining a language)

A program is a sequence of symbols that come from an alphabet

This sequence is called a string

A language is a set of strings whose symbols come from an alphabet

This set might be infinite

A legal program of a language is a string contained in the set of strings of the language
Noam Chomsky

Noam Chomsky (1959) defined four classes of languages. His main interest was natural languages but his ideas are applicable to computer languages too.

He proposed defining a language as the set of all possible strings that can be generated by applying a set of rewriting rules or productions to a starting "goal" symbol.

A rewriting rule or production allows one to replace a given string of symbols with another string of symbols.

The set of symbols is augmented to contain not only the terminal symbols (the symbols in the original alphabet) but also nonterminal symbols used by the rewrite rules.

Rules for producing an identifier in C:

```
IDENT → LETTER, LETTERORDIGITS
LETTER → _
LETTER → A
LETTER → B
...
LETTER → Z
...
LETTER → z
DIGIT → 0
...
DIGIT → 9
LETTERORDIGITS → LETTERORDIGIT, LETTERORDIGITS
LETTERORDIGITS →
LETTERORDIGIT → LETTER
LETTERORDIGIT → DIGIT
```
Types of Chomsky Grammars

Noam Chomsky defined four types of languages

Type 0 - Unrestricted grammars
Any number of symbols may appear on both sides of the arrow

Type 1 - Context-sensitive grammars
The number of symbols on the left side of the arrow may not exceed the number of symbols on the right side of the arrow

Type 2 - Context-free grammars
There must be exactly one symbol on the left side of the arrow

Type 3 - Regular grammars
There must be exactly one symbol on the left side of the arrow
The right side must be a terminal symbol followed by an optional nonterminal symbol

Each type is contained in the previous type
Lower numbered types define more complicated languages
Type 0 - Unrestricted grammars

Any number of symbols may appear on both sides of the arrow.

These grammars can recognize any computable function.

Most properties of a type 0 grammar (such as "are there any strings in the language?") are undecidable.
Type 1 - Context-sensitive grammars

The number of symbols on the left side of the arrow may not exceed the number of symbols on the right side of the arrow.

Generally too complicated for practical applications.
Type 2 - Context-free grammars

There must be exactly one symbol on the left side of the arrow

These languages are the same as the BNF languages

Many languages define their syntax as a context-free language

Easy to require balancing of parentheses, arithmetic expressions, if statements etc.

Cannot define the requirement that identifiers must be defined before use

These languages can be recognized using a stack (if input can be reread)

Many languages use grammars that do not require backing up the input
Type 3 - Regular grammars

There must be exactly one symbol on the left side of the arrow. The right side must be a terminal symbol followed by an optional nonterminal symbol.

This language is identical to the language defined by classical regular expressions.

They can be recognized by a finite-state machine using one pass through the string.

Regular grammars are not powerful enough for most computer languages but are useful for describing the components of a language such as identifiers, numbers, comments etc.
Backus Naur form (BNF)

This description of a language was developed for describing ALGOL 60

After the publication of the ALGOL 60 report most languages were described with a similar technique Identical to Chomsky type 2 or context free languages

The terms are now used interchangeably

Original Notation

 ::= used for →
<nameOfNonterminal> used for nonterminals
| for alternatives
. at the end of a production

Example

 <conditional statement> ::= 
     if <Boolean expression> then <statement> else <statement>
     | if <Boolean expression> then <statement>.

C++ syntax (from the standard)
(a different font is used for terminals)

statement:
    labeled-statement
    expression-statement
    compound-statement
    selection-statement
    iteration-statement
    jump-statement
    declaration-statement
    try-block

selection-statement:
    if ( condition ) statement
    if ( condition ) statement else statement
switch (condition) statement
Advantages of BNF for describing computer languages

Precision
Can attach semantics to syntax productions
Can (almost) automatically construct compilers from the production rules
Parse Trees

Given a set of productions describing a language and a string in the language a parse tree is a visual proof that the string is in the language.

For example, the following productions generate all strings containing balanced parentheses:

\[ S \rightarrow SS \mid (S) \mid () \]

How do we prove that \((())()()\) is legal?

\[ S \Rightarrow SS \Rightarrow (S)S \Rightarrow (S)S \Rightarrow (()S)S \Rightarrow (()()S) \Rightarrow (()())() \]

Or as a parse tree:

```
          S
         / \    |
        S   S   |
       / \    |
      ( S ) ( )
     /     |
    S      S
   /   |
  ( ) ( )
```

Note that every node is either a terminal symbol or has descendants that correspond to a production of the node’s nonterminal symbol.
Ambiguity

It is possible that a legal string in a language has more than one parse tree

Consider the simple grammar for arithmetic expressions

\[ e \rightarrow n \mid e + e \mid e * e \]
\[ n \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \]

Then the expression

\[ 2 + 3 * 4 \]

might be parsed

\[ (2 + 3) * 4 \]
or

\[ 2 + (3 * 4) \]

(This happens in English too!)

If this happens then the grammar is said to be ambiguous and strings in the language might have more than one interpretation

An example in programming languages is the "dangling else"

\[ \text{if A then if B then C else D} \]

Is this

\[ \text{if A then (if B then C else D)} \]
or

\[ \text{if A then (if B then C) else D} \]

Sometimes it is possible to rewrite the grammar productions to eliminate ambiguity
Java's grammar for the if statement

Statement:
  StatementWithoutTrailingSubstatement
  IfThenStatement
  IfThenElseStatement
  WhileStatement
  ForStatement

StatementWithoutTrailingSubstatement:
  ExpressionStatement
  ContinueStatement
  ThrowStatement

StatementNoShortIf:
  StatementWithoutTrailingSubstatement
  IfThenElseStatementNoShortIf
  WhileStatementNoShortIf
  ForStatementNoShortIf

IfThenStatement:
  if ( Expression ) Statement

IfThenElseStatement:
  if ( Expression ) StatementNoShortIf else Statement

IfThenElseStatementNoShortIf:
  if ( Expression ) StatementNoShortIf else StatementNoShortIf

WhileStatement:
  while ( Expression ) Statement

WhileStatementNoShortIf:
  while ( Expression ) StatementNoShortIf

ForStatement:
  for ( ForInit_opt ; Expression_opt ; ForUpdate_opt ) Statement

ForStatementNoShortIf:
  for ( ForInit_opt ; Expression_opt ; ForUpdate_opt ) StatementNoShortIf
Writing Parsers and Compilers

It is possible to write a BNF grammar for a language that parses the individual characters of a program.

Using the machinery of the full parser to parse numbers, identifiers etc. can be slow.

Generally, parsing a language is divided into lexical parsing and syntactic parsing.

The lexical parser finds the tokens of the language such as identifiers, operators, strings, comments, and categorizes these tokens into groups.

Generally the meaning of a program does not change if the characters of a literal string are changed.

The syntactic parser then parses a language defined over an alphabet of tokens.

This way the syntactic parser does not have to concern itself with the details of number representation and conversion etc.

After several compilers were written (and especially after T. E. Irons wrote a syntax directed compiler) it was realized that there were several recurring algorithms that were common to many compilers.

The programs lex and yacc were written to automate the common parts of compiler writing.

These programs can be considered to be "compiler compilers", i.e., programs to create compilers.
Parsing Strategies

Given a grammar - say a BNF grammar - how do we parse a string?

We could start with the goal symbol and try to expand it into the given string

This is called top-down recursive descent parsing

We could start with the first symbol in the input string and find productions that have this symbol first and try to continually replace the first symbols in the string with a "higher" nonterminal symbol

If we replace the entire string with the goal symbol then we are done

This would be a bottom-up parser

For some grammars (called LR(k) grammars) we only need to look ahead k symbols to decide which production to use

For these grammars a parsing strategy can be devised that is fast and never backs up

This is the kind of grammar that yacc and bison require

We will be using these grammars and tools in lab
Top-Down Recursive Descent Parsers

A general top-down parser can be written by

Writing a procedure for every nonterminal symbol in the grammar that parses the next symbols in the input stream as that nonterminal or reports failure

Then calling the procedure for the goal symbol will parse the input string

For a production like

\[ A \rightarrow B \mathord{C} \mid D \mathord{E} \mathord{F} \]

we can write the procedure

\[
\begin{align*}
\text{pos} &= \text{place in string} \\
\text{if} \ ( B() ) \ \text{then if} \ ( C() ) \ \text{return success} \\
& \ \text{restore string position from pos} \\
\text{if} \ ( D() ) \ \text{then if} \ ( E() \ \&\& \ F() ) \ \text{return success} \\
& \ \text{restore string position from pos} \\
& \ \text{return failure}
\end{align*}
\]

A terminal is matched by checking the next symbol in the input string for a match and advancing the position if there is a match else reporting failure

If the grammar is nice enough we never have to back up the input stream

This technique is good for many simple parsing tasks
How would we write a top-down parser for arithmetic expressions?

x
(x)
x+x
x*x
(x+x)*(x)+x
Top-down Grammar for Arithmetic Expressions

(terminal symbols are underlined)

<empty> always succeeds and matches nothing

primary → x | ( expression )
product → primary product-tail
product-tail → * primary | <empty>
sum → product sum-tail
sum-tail → + sum | <empty>
expression → sum

Although very fast top-down parsers can be made, the grammar may not be too "natural"
Bottom-Up Parsers

The idea in a bottom-up parser is to continually replace symbols near the beginning of a string with a nonterminal that can produce the replaced symbols.

This would involve a lot of string concatenations etc.

If we use a stack for storing all of the symbols that have been scanned (and sometimes replaced) we never have to modify the input string and the effective string we are considering is the concatenation of the stack symbols with the rest of the input string.

We can preprocess the input grammar to make a table of possible reductions keyed on the next symbol to scan and the contents of the stack.

Then a parsing step is either
- push the next symbol onto the stack (shift)
- replace some symbols on the top of the stack with a nonterminal (reduce)

Sometimes there are ambiguities in the grammar.

With some grammars it is impossible to decide whether to shift another symbol onto the stack or reduce the top symbols already on the stack without looking ahead an unbounded number of symbols.

Guessing wrong may cause the parse to fail.

There might be two possible reductions that can be made at some point.

Most parser generators indicate these problems although there may be a default action taken if the warnings are ignored.
Semantic Analysis

It is not enough to know that a string is a valid program - we need to know what it means.

The meaning of a program can be indicated by attaching information to each node of the parse tree.

For example, the information on the parse tree node for the multiplication operator (*) can be either:
- the result of multiplying the value of its two child nodes,
- the computer code for calculating the product of the child nodes using the child node values.

The semantic value of a node in the parse tree generally depends on the values of the nodes below it in the tree.

The semantic values on the nodes can be calculated after the parse tree is generated or while it is being built up.

Modern compilers do both and "walk" the parse tree several times to modify it and calculate values.