What this course is

- A study of programming language paradigms
  - Imperative
  - Functional
  - Logical
  - Object-oriented

- Tour of programming language history & roots.
- Introduction to core language design & implementation concepts.
- Exposure to languages/paradigms you may not have used before.
- Reasoning about language benefits/pitfalls.
- Exploration of implementation issues.
- Understanding and appreciation of language standards.
- Ability to more quickly learn new languages.
What this course isn’t

- A comprehensive study of certain languages.
- An exercise in learning as many languages as possible.
- A software engineering course.
- A compiler course.
Introduction

The main themes of programming language design and use:

- Paradigm (model of computation)
- Expressiveness
  - control structures
  - abstraction mechanisms
  - types and their operations
  - tools for programming in the large
- Ease of use: writeability / readability / maintainability
Language as a tool for thought

- Role of language as a communication vehicle among programmers can be just as important as ease of writing.
- All general-purpose languages are *Turing complete* (They can compute the same things).
- But languages can make expression of certain algorithms difficult or easy.
  - Try multiplying two Roman numerals.
- Idioms in language A may be useful inspiration when writing in language B.
Idioms

- Copying a string \( q \) to \( p \) in C:
  
  ```c
  while (*p++ = *q++) ;
  ```

- Removing duplicates from the list @xs in Perl:
  
  ```perl
  my %seen = ();
  @xs = grep { ! $seen{$_}++; } @xs;
  ```

- Computing the sum of numbers in list \( xs \) in Haskell:
  
  ```haskell
  foldr (+) 0 xs
  ```

Is this natural?  

*It is if you’re used to it*
Programming paradigms

■ **Imperative (von Neumann):** Fortran, Pascal, C, Ada
  ◆ programs have mutable storage (state) modified by assignments
  ◆ the most common and familiar paradigm

■ **Functional (applicative):** Scheme, Lisp, ML, Haskell
  ◆ functions are first-class values
  ◆ *side effects* (e.g., assignments) discouraged

■ **Logical (declarative):** Prolog, Mercury
  ◆ programs are sets of assertions and rules

■ **Object-Oriented:** Simula 67, Smalltalk, C++, Ada95, Java, C#
  ◆ data structures and their operations are bundled together
  ◆ inheritance

■ Functional + Logical: Curry
■ Functional + Object-Oriented: O’Caml, O’Haskell
Before FORTRAN/COBOL/ALGOL, programs were written in assembly.

FORTRAN

- Invented by John Backus et al., released in 1957.
- First successful high-level programming language.
- Primary use: scientific computing and mathematics.
- Example:

  \[ A = C + D \]

COBOL

- Designed by committee, released late 1960.
- Common or Business-Oriented Language.
- Data processing, business, finance, administrative systems.
- Introduced structures.
- Example:

  `ADD C TO D GIVING A`
Beginnings (continued)

- **ALGOL**
  - Invented by a group of European & American computer scientists, released in 1958.
  - Popularized many PL concepts still in use today.
    - BNF
    - Compound statements using blocks
    - case statement
    - Call-by-reference
    - Concurrency
    - Orthogonality
  - Was not a commercial success (e.g., no standard I/O).

```
IF Ivar > Jvar THEN Ivar ELSE Jvar FI := 3;
```
Genealogy

- FORTRAN (1957) ⇒ Fortran90, HP
- COBOL (1960) ⇒ COBOL 2000
- Algol60 ⇒ Algol68/Algol W ⇒ Pascal ⇒ Ada
- Algol60 ⇒ BCPL ⇒ C ⇒ C++
- Algol60 ⇒ Simula ⇒ Smalltalk
- APL ⇒ J
- Snobol ⇒ Icon
- Lisp ⇒ Scheme ⇒ ML ⇒ Haskell

with lots of cross-pollination: e.g., Java is influenced by C++, Smalltalk, Lisp, Ada, etc.
High vs. low level languages

- Low-level languages mirror the physical machine:
  - Assembly, C, Fortran
- High-level languages model an abstract machine with useful capabilities:
  - ML, Setl, Prolog, SQL, Haskell
- Wide-spectrum languages try to do both:
  - Ada, C++, Java, C#
- High-level languages have garbage collection, are often interpreted, and cannot be used for real-time programming. The higher the level, the harder it is to determine cost of operations.
Common ideas

Modern imperative languages (e.g., Ada, C++, Java) have similar characteristics:

- large number of features (grammar with several hundred productions, 500 page reference manuals, . . .)
- a complex type system
- procedural mechanisms
- object-oriented facilities
- abstraction mechanisms, with information hiding
- several storage-allocation mechanisms
- facilities for concurrent programming
- facilities for generic programming
Language standards

Developed by working groups of standards bodies (ANSI, ISO).

- Main goal: defines one language, increases portability.
- Pro: Discourages countless language flavors (like LISP)
- Con: Places creative freedom in the hands of a few people.
- Major compiler manufacturers generally align to the standards.
- Defines syntactic and semantic correctness (sometimes partially).
- Enforcement is often left to individual compiler implementations.

Example: incorrect code, but GNU C++ compiler doesn’t warn by default:

```c++
int x;
int y = x + 2;  // x is undefined
```
Language libraries

The programming environment may be larger than the language.

- The predefined libraries are *indispensable* to the proper use of the language, *and its popularity*.
- The libraries are defined in the language itself, but they have to be internalized by a good programmer.

Examples:

- C++ standard template library
- Java Swing classes
- Ada I/O packages
Syntax and semantics

- Syntax refers to external representation:
  - Given some text, is it a well-formed program?

- Semantics denotes meaning:
  - Given a well-formed program, what does it mean?
  - Often depends on context (e.g. C++ keyword `const`).

The division is somewhat arbitrary.

- Note: It *is* possible to fully describe the syntax and semantics of a programming language by syntactic means (e.g., Algol68 and W-grammars), but this is highly impractical.

  Typically use a grammar for the context-free aspects, and different method for the rest.

- Similar looking constructs in different languages often have subtly (or not-so-subtly) different meanings
Compilation overview

Major phases of a compiler:

1. lexer: text $\rightarrow$ tokens
2. parser: tokens $\rightarrow$ parse tree
3. semantic analyzer: parse tree $\rightarrow$ abstract syntax tree
4. intermediate code generation
5. optimization (machine independent): local & global redundancy elimination, loop optimization
6. target code generation
7. optimization (machine dependent): instruction scheduling, register allocation, peephole optimization
Grammars

A grammar $G$ is a tuple $(\Sigma, N, S, \delta)$

- $\Sigma$ is the set of *terminal* symbols (alphabet)
- $N$ is the set of *non-terminal* symbols
- $S$ is the distinguished non-terminal: the root symbol
- $\delta$ is the set of rewrite rules (productions) of the form:

  \[
  ABC \ldots ::= XYZ \ldots
  \]

  where A, B, C, X, Y, Z are terminals and non terminals.

- The *language* is the set of sentences containing *only* terminal symbols that can be generated by applying the rewriting rules starting from the root symbol (let’s call such sentences *strings*)
BNF for context-free grammars

(BNF = Backus-Naur Form) Some conventional abbreviations:

- **alternation**: `Symb ::= Letter | Digit`
- **repetition**: `Id ::= Letter {Symb}`
  - or we can use a Kleene star: `Id ::= Letter Symb*`
  - for one or more repetitions: `Int ::= Digit+`
- **option**: `Num ::= Digit+[. Digit*]`
- abbreviations do not add to expressive power of grammar
- need convention for metasymbols – what if “|” is in the language?
<typedef> ::= type <typedeflist>
<typedeflist> ::= <typedef> [ <typedeflist> ]
<typedef> ::= <typeid> = <typespec> ;
<typespec> ::= <typeid> |
       <arraydef> | <ptrdef> | <rangedef> | <enumdef> | <recdef>
<typeid> ::= <ident>
<arraydef> ::= [ packed ] array ‘[’ <rangedef> ‘]’ of <typeid>
<ptrdef> ::= ^ <typeid>
<rangedef> ::= <number> .. <number>
<number> ::= <digit> [ <number> ]
<enumdef> ::= ( <idlist> )
<idlist> ::= <ident> { , <ident> }
<recdef> ::= record <vardecllist> end ;
The Chomsky hierarchy

- Regular grammars (Type 3)
  - all productions can be written in the form: N ::= TN
  - one non-terminal on left side; at most one on right

- Context-free grammars (Type 2)
  - all productions can be written in the form: N ::= XYZ
  - one non-terminal on the left-hand side; mixture on right

- Context-sensitive grammars (Type 1)
  - number of symbols on the left is no greater than on the right
  - no production shrinks the size of the sentential form

- Type-0 grammars
  - no restrictions
Regular expressions

Regular expressions can be used to generate or recognize regular languages. We say that a regular expression \( R \) denotes the language \([R]\).

Basic regular expressions:

- \( \epsilon \) denotes \( \emptyset \)
- a character \( x \), where \( x \in \Sigma \), denotes \( \{x\} \)
- (sequencing) a sequence of two regular expressions \( RS \) denotes \( \{\alpha\beta | \alpha \in \[R\], \beta \in \[S]\}\}
- (alternation) \( R|S \) denotes \( \[R\] \cup \[S]\)
- (Kleene star) \( R^* \) denotes the set of strings which are concatenations of zero or more strings from \( \[R\] \)
- parentheses are used for grouping

Shorthands:

- \( R? \equiv \epsilon|R \)
- \( R^+ \equiv RR^* \)
Regular grammar example

A grammar for floating point numbers:

\[
\text{Float ::= Digits | Digits . Digits} \\
\text{Digits ::= Digit | Digit Digits} \\
\text{Digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9}
\]

A regular expression for floating point numbers:

\[
(0|1|2|3|4|5|6|7|8|9)^+ (0|1|2|3|4|5|6|7|8|9)^? 
\]

Perl offer some shorthands:

\[
[0-9]+(\ . [0-9]+)?
\]
or

\[
\d+(\ . \d+)?
\]
Lexical Issues

Lexical: formation of words or tokens.

- Described (mainly) by regular grammars
- Terminals are characters. Some choices:
  - character set: ASCII, Latin-1, ISO646, Unicode, etc.
  - is case significant?
- Is indentation significant?
  - Python, Occam, Haskell

Example: identifiers

\[
\begin{align*}
\text{Id} & ::= \text{Letter} \ \text{IdRest} \\
\text{IdRest} & ::= \epsilon \ | \ \text{Letter} \ \text{IdRest} \ | \ \text{Digit} \ \text{IdRest}
\end{align*}
\]

Missing from above grammar: limit of identifier length
Parse trees

A parse tree describes the grammatical structure of a sentence

- root of tree is root symbol of grammar
- leaf nodes are terminal symbols
- internal nodes are non-terminal symbols
- an internal node and its descendants correspond to some production for that non terminal
- top-down tree traversal represents the process of generating the given sentence from the grammar
- construction of tree from sentence is parsing
Ambiguity

If the parse tree for a sentence is not unique, the grammar is ambiguous:

\[ E ::= E + E \mid E \ast E \mid \text{Id} \]

Two possible parse trees for “A + B \ast C”:

- ((A + B) \ast C)
- (A + (B \ast C))

One solution: rearrange grammar:

\[ E ::= E + T \mid T \]
\[ T ::= T \ast \text{Id} \mid \text{Id} \]

Harder problems – disambiguate these (courtesy of Ada):

- function_call ::= name (expression_list)
- indexed_component ::= name (index_list)
- type_conversion ::= name (expression)
Dangling else problem

Consider:

\[
\begin{align*}
S &::= \text{if } E \text{ then } S \\
S &::= \text{if } E \text{ then } S \text{ else } S
\end{align*}
\]

The sentence

\[
\text{if } E_1 \text{ then if } E_2 \text{ then } S_1 \text{ else } S_2
\]

is ambiguous (Which \text{then} does \text{else } S_2 \text{ match}?)

Solutions:

- Pascal rule: else matches most recent if
- grammatical solution: different productions for balanced and unbalanced if-statements
- grammatical solution: introduce explicit end-marker

The general ambiguity problem is unsolvable
Scanners and parsers

- **Scanners** (or *tokenizers*) read input, identify, and extract small input fragments called tokens.
  - Identifiers
  - Constants
  - Keywords
  - Symbols: (, ), [, ], !, =, !=, etc.

- **Parsers** accept tokens and attempt to construct a parse tree.
  - **LL** (or: recursive descent, predictive) parsers are depth-first, begin at the start symbol, predict the next rewrite rule, and recurse on it. Implementation: “by hand” or table-driven.
  - **LR** (or: bottom-up) parsers find LHS non-terminals that match the input tokens already seen. Normally faster in production compilers (exception: gcc). Implementation: almost always table-driven.

Relationships: **LL ⊂ LR ⊂ CF**
LL Parsers

- LL stands for: left-to-right, leftmost derivation.
- Also known as top-down, recursive descent, or predictive parsers.
- Begin at the root symbol.
- For each RHS non-terminal, decide which rewrite rule to use (if more than one).
- Decision: “predict” the next input tokens, pick a rewrite rule.
- Ideal: only one rule to choose from. Deterministic.
- Error: no rule exists.
- Resolving nondeterminism: “look ahead” to beyond the next token.
- We can look ahead an arbitrary number of tokens. Call this number $k$.
- We refer to parsers with $k$ lookahead as LL($k$) parsers.
- Note 1: it may be possible to modify the grammar to an equivalent grammar with a smaller (or no) lookahead requirement.
- Note 2: most production LL parsers are LL(1).
Problems with LL parsing

**Left recursion:** a grammar is left-recursive if there exists non-terminal $A$ such that $A \Rightarrow^+ A\alpha$ for some $\alpha$. Example:

$$\begin{align*}
id\_list & \quad \Rightarrow id\_list\_prefix\ ; \\
id\_list\_prefix & \quad \Rightarrow id\_list\_prefix\ ,\ id \\
& \quad \Rightarrow id
\end{align*}$$

**Common prefixes:** if there exists a non-terminal $A$ and terminal $b$ such that there exist rules $R_1 : A \Rightarrow^* b\ldots$ and $R_2 : A \Rightarrow^* b\ldots$

$$\begin{align*}
stmt & \quad \Rightarrow id := expr \\
& \quad \Rightarrow id\ (\ argument\_list )
\end{align*}$$

Solution to both issues: rewrite the grammar.

(Examples courtesy of Scott. See p.84 for solutions).
LR Parsers

- Stands for: left-to-right, rightmost derivation.
- Also known as bottom up or shift-reduce parsers.
- With LR parsers, the bottom of the parse tree is built first.
- If the root symbol is reachable, the parser accepts the input.
- Main data structure is a stack.
- Main operations are: **shift** and **reduce**.
- LR parsers **shift** tokens to the stack.
- Each time a token is shifted, the stack is checked to see if the tokens match the right side of a rule.
- If so, we replace the tokens on the stack with the left-hand side of the rule. This is the **reduce** step.
- If the stack is empty when the input is fully read, the input is accepted.
- If the stack is non-empty, there exist tokens with no corresponding rule: error.
Problems with LR parsing

Shift-reduce conflicts: when the choice of shifting or reducing is non-deterministic. Example:

\[
\text{if stmt} \quad \Rightarrow \quad \text{IF expr THEN stmt} \\
\quad \Rightarrow \quad \text{IF expr THEN stmt ELSE stmt}
\]

Suppose the parser has read \text{IF expr THEN stmt} (but not yet ELSE) The parser could shift ELSE or reduce the above to if\_stmt—there isn’t enough information for it to properly choose.

Solutions: rewrite grammar (see previous slides), introduce lookahead.
Creating scanners and parsers

- **Lex** (or **Flex**) is a lexical analyzer generator.
  - Input: rules containing regular expressions.
  - Output: C code. Can be compiled into a standalone lexical analyzer or integrated into a parser.

- **Yacc** (or **Bison**) is a parser generator.
  - Input: Context-free grammar and Lex generated source code (optional).
  - Output: An LR parser.