CS558 Programming Languages

Goals of the course:
- Learn fundamental structure of programming languages.
- Understand key issues in language design and implementation.
- Be aware of the range of available languages and their uses.
- Learn how to learn a new language.

Method of the course:
- Fairly conventional survey textbook, with emphasis on implementation issues.
- Homework exercises involve programming problems in real languages.
- Most homework problems will involve modifying implementations of “toy” languages that illustrate key features and issues.
- Exercises will use (at least) three languages: C, Java, and Standard ML.
- Among them, these languages illustrate most of the important concepts in current language designs.

Non-goals
- Teaching how to program.
- Teaching how to write programs in any particular language(s).
- Surveying/cataloging the features of lots of different languages.
- Comprehensive coverage of programming paradigms (e.g., will skip logic and concurrent programming material).
- Will mostly be concerned with interpreting abstract syntax for the toy languages, and will spend only a little bit of time on parsing and code generation. (Not a compiler course!)

Some Languages

What languages do you know?

Don’t forget things like:
awk, perl, tcl
SQL, other database query languages.
spreadsheet expression languages
text processing languages, tex, troff
application-specific languages.
“Higher-Level” Programming Languages

Consider a simple (dumb) algorithm for testing primality.

In Java:

```java
public static boolean isPrime (int n)
int d;
for (d = 2; d < n; d++)
if (n % d == 0)
    return false;
return true;
```

In C: very similar to Java, except that boolean is represented by int.

In Standard ML (using a recursive function):

```ml
fun isPrime (n:int) : bool =
    let fun no_divisor (d:int) : bool =
        (d >= n) orelse
        ((n mod d <> 0) andalso
        (no_divisor (d+1)))
    in no_divisor 2
    end
```

In Intel X86 assembler:

```assembly
.globl isPrime
isPrime:
pushl %ebp ; set up procedure entry
movl %esp,%ebp
pushl %esi
pushl %ebx
movl 8(%ebp),%ebx ; fetch arg n from stack
movl $2,%esi ; set divisor d := 2
cmpl %ebx,%esi ; compare n,d
jge true ; jump if d >= n
loop:
movl %ebx,%eax ; set n into ....
cltd ; ... dividend register
idivl %esi ; divide by d
testl %edx,%edx ; remainder 0?
jne next ; jump if remainder non-0
xorl %eax,%eax ; set ret value := false(0)
jmp done
next:
    incl %esi ; increment d
cmpl %ebx,%esi ; compare n,d
    jl loop ; jump if d < n
true:
movl $1,%eax ; set ret value := true(1)
done:
    leal -8(%ebp),%esp ; clean up and exit
    popl %ebx
    popl %esi
    leave
    ret
```

High-level Languages: General Characteristics

- Complex Expressions (Arithmetic, Logical, ...)
- Structured Control Operators (Loops, Conditionals, Cases)
- Composite Types (Arrays, Records, etc.)
- Type Declarations and Type Checking
- Multiple storage classes (global/local/heap)
- Procedures/Functions, with private scope, maybe first-class
- Maybe high-level control mechanisms (Exceptions, Back-tracking, etc.)
- Maybe abstract data types, modules, objects, etc.

Machine Code Characteristics

- Explicit registers for values and intermediate results.
- Low-level machine instructions to implement operations.
- Control flow based on labels and conditional branches.
- Explicit memory management (e.g., stack management for procedures).

Programming paradigms

Imperative (including object-oriented)

- Conventional
- Statement-oriented
- “Stateful”

Functional

- Expression-oriented
- No side effects
- Organized into functions

Logic

- Predicate-oriented
- Organized into relations

Concurrent/Parallel

- Multiple threads of control
- Synchronization
Programming Contexts

Programming “in the Small”
- Expressions
- Structured Control Flow
- Structured Data

Programming “in the Large”
- Modules and Separate Compilation
- Code Re-use; Polymorphism
- Object-oriented Programming

Programming with Types
- Security
- Structure

C
- Procedural imperative language.
- Widely used for systems programming, especially for Unix/Linux.
- First language to be implemented on almost any new machine.
- Influential expression-oriented syntax.
- Supports “bit-twiddling” low-level manipulation.
- Generates efficient, predictable machine code.
- Weak type system; run-time crashes common.
- Weak structuring mechanisms.

Java
- Object-oriented imperative language.
- Hyped for “web programming.”
- Designed as a “better C++.”
- Core language largely derived from C, C++.
- Strong static type-checking.
- Classes serve as primary structuring mechanism.
- Automatic heap allocation and garbage collection.
- Extensive libraries.
- Portable interpretive environment (JVM).

Standard ML
- Primarily functional language.
- Designed for symbolic manipulation; inherits from LISP, Scheme.
- Strong, static type inference.
- Simple, uniform, powerful recursive datatype mechanism.
- Recursion is primary control structure.
- Powerful parameterized module mechanism.
- Automatic heap allocation and garbage collection.
- Top-level “read-eval-print” loop for development.
Language Description and Documentation

For programmers, compiler-writers, and students . . .

Syntax (Easy)
- Grammars; BNF and Syntax Charts

Semantics (Hard)
- Informal
- Formal: Operational, Denotational, Axiomatic

Learning about a Language
- Reference Manuals
- User Guides
- Texts and tutorials

Grammars
- Used for description, parsing, analysis, etc.
- Based on recursive definition of program structure.
- Rich theory with connections to automatic parser generation, push-down automata, etc.
- Many possible representations, including BNF (Backus-Naur Form), EBNF (Extended BNF), syntax charts, etc.

BNF Example

```
<program> ::= BEGIN <statement-seq> END
<statement-seq> ::= <statement>  
<statement> ::= <for-statement>  
<for-statement> ::= WHILE <expression>  
<expression> ::= <factor>  
<factor> ::= <variable>  
```

Formal Grammars

A (Context-free) Grammar over a given character set consists of
- A set of terminals, which are strings of zero or more characters.
- A set of nonterminals, which are variables representing a set of terminals.
- A set of productions, each of which has a left side consisting of a single nonterminal and a right side consisting of zero or more terminals or nonterminals.
- A distinguished starting nonterminal.

We can apply a production to a string by replacing some instance of the left side nonterminal by the right side.

The context-free language $L(G)$ generated by a grammar $G$ is the set of character strings that can be derived from the starting nonterminal by applying productions—in any order—until no nonterminals remain.

Simple Example

Character set: \{ (, ) \}
Terminals: \{ (, ) \}

Productions:

```
S ::= (S)  
S ::= SS  
S ::= \epsilon (the empty string)  
```

Starting nonterminal: $S$

Sample derivation:

```
S \rightarrow (S) \rightarrow (SS) \rightarrow ((S)S) \rightarrow (())S \rightarrow (())(S) \rightarrow (())())  
```

This grammar generates the language of strings of properly matched parentheses.

It is often useful to think of a derivation as a tree.
BNF and EBNF

BNF was invented ca. 1960 and used in the formal description of Algol-60. It is just a particular notation for grammars, in which

- Nonterminals are represented by names inside angle brackets, e.g., <program>, <expression>, <S>.
- Terminals are represented by themselves, e.g., WHILE, , 3. The empty string is written as <empty>.

EBNF is (any) extension of BNF, usually with these features:

- A vertical bar, |, represents a choice,
- Parentheses, ( and ), represent grouping,
- Square brackets, [ and ], represent an optional construct,
- Curly braces, { and }, represent zero or more repetitions,
- Nonterminals begin with upper-case letters.
- Non-alphabetic terminal symbols are quoted, at least when necessary to avoid confusion with the meta-symbols above.

EBNF Example

\[
\begin{align*}
\text{Program} & ::= \text{BEGIN Statement-seq END} \\
\text{Statement-seq} & ::= \text{Statement} \\
& \quad | \quad [ \text{'} \text{'} \text{Statement-seq} \text{'} \text{'} ] \\
\text{Statement} & ::= \{ \text{While-statement} \mid \text{For-statement} \} \\
\text{While-statement} & ::= \text{WHILE Expression} \text{ DO Statement-seq END} \\
\text{Expression} & ::= \text{Factor} \{ \text{ (AND} \mid \text{OR) Factor} \} \\
\text{Factor} & ::= \{ \text{'} \text{'} \text{Expression} \text{'} \text{'} \mid \text{Variable} \\
\text{For-statement} & ::= \ldots \\
\text{Variable} & ::= \ldots
\end{align*}
\]

Syntax Analysis (Parsing)

Parser recognizes syntactically legal programs (as defined by a grammar) and rejects illegal ones.

- Successful parse also captures hierarchical structure of programs (expressions, blocks, etc.).
- Convenient representation for further semantic checking (e.g., typechecking) and for code generation.
- Failed parse provides error feedback to the user indicating where and why the input was illegal.

Any context-free language can be parsed by a computer program, but only some can be parsed efficiently.

- Two general approaches to parsing are “top-down” and “bottom-up”.
- Parsers are often written by hand, but there are well-established tools (parser generators) that create parsers automatically from grammar descriptions.
- Hand-written parsers are almost always top-down; we’ll only study these.

Lexical Analysis

Parsing usually works on a stream of tokens representing terminals in the grammar.

Converting program text into token stream is job of lexical analyzer.

- Detect and identify keywords and identifiers.
- Convert multi-character symbols into single tokens.
- Handle numeric and string literals.
- Remove whitespace and comments.

Regular expressions and finite automata are a nice basis for formalizing what tokens look like and automating the lexical analyzer.

- We won’t study here (but see text).

In practice, lexical analyzers are often written by hand instead, to achieve high performance.

- Can be slowest part of compiler (because it looks at every character of the program text)!
Parse Trees

Graphical representation of a derivation.

Given this grammar:

\[
expr \rightarrow expr + expr \mid expr \ast expr \\
\qquad \mid (expr) \mid -expr \mid id
\]

Example tree for derivation of sentence \(-(id + id)\):

```
expr
  `- expr
      `- expr
          `( expr )
              `expr + expr
                  `id
                      `id
```

Each application of a production corresponds to an **internal** node, labeled with a **non-terminal**.

Leaves are labeled with **terminals**.

The derived sentence is found by reading leaves left-to-right.

Ambiguity

A given sentence in \(L(G)\) can have more than one parse tree. Grammars \(G\) for which this is true are called *ambiguous*.

Example: with our grammar, the sentence

\[id + id \ast id\]

has two parse trees:

```
expr
  `- expr
      `- expr
          `( expr )
              `expr + expr
                  `id
                      `id
```

We may think of the left tree as being the “correct” one, but nothing in the grammar says this.

To avoid the problems of ambiguity, we can:

- Rewrite grammar
- Use “disambiguating rules” when we implement parser for grammar.

A classic ambiguity: the “dangling else”

Suppose we want else clauses to be optional in if statements.

Here’s a possible grammar:

\[
stmt \rightarrow if \ expr \ then \ stmt \\
\quad \mid if \ expr \ then \ stmt \ else \ stmt \mid ...
\]

But with this grammar the sentence

\[if \ E_1 \ then \ if \ E_2 \ then \ S_1 \ else \ S_2\]

has two possible parse trees:

```
stmt
  `if
      `expr
          `then
              stmnt
                  `if
                      `expr
                          `then
                              stmnt
                                  `else
                                      stmnt
                                          `if
                                              `expr
                                                  `then
                                                      stmnt
                                                          `else
                                                              stmnt
```

Resolving Ambiguity by Rewriting Grammar

Usually want the first tree (else goes with most recent then), but grammar is ambiguous.

Solution: rewrite grammar using new non-terminals \(mst\) (“matched statement”) and \(ust\) (“unmatched statement”).

\[
stmt \rightarrow mst \mid ust \\
\quad \mid if \ expr \ then \ mst \ else \ mst \\
\quad \mid ... \\
\quad \mid if \ expr \ then \ stmt \\
\quad \mid if \ expr \ then \ mst \ else \ ust
\]

Now only one parse is possible. Assuming \(S_1, S_2\) are not unmatched if statements:

```
stmt
  `ust
      `if
          `expr
              `then
                  stmnt
                      `if
                          `expr
                              `then
                                  stmnt
                                      `else
                                          stmnt
                                              `if
                                                  `expr
                                                      `then
                                                          stmnt
                                                              `else
                                                                  stmnt
```
Ambiguity in Arithmetic Expressions

A grammar such as
\[ E \rightarrow E + E \mid E - E \mid E \ast E \mid E / E \]
\[ E \uparrow E \mid (E) \mid - E \mid \text{id} \]
is ambiguous about order of operations.

Want to define
- **Precedence** - which operation is done first?
- **Associativity** - is \( X \ast Y \ast Z \) equivalent to \((X \ast Y) \ast Z \) or to \( X \ast (Y \ast Z) \) (assuming \( \ast \) and \( \ast \) have same precedence).

The “usual” rules (based on common usage in written math) give the following precedences, highest first:
- (unary minus)
- (exponentiation)
- \(/\)
- \(+\)

All the binary operators are left-associative except exponentiation (\(\uparrow\)).

We can handle precedence/associativity information as “side-conditions” to ambiguous grammar when building a parser (by hand or via a parser generator).

Rewriting Arithmetic Grammars

Can build precedence/associativity into grammar using extra non-terminals, e.g.

\[ \text{atom} \rightarrow \text{expr} \mid \text{id} \]
\[ \text{primary} \rightarrow -\text{primary} \mid \text{atom} \]
\[ \text{factor} \rightarrow \text{primary} \uparrow \text{factor} \mid \text{primary} \]
\[ \text{term} \rightarrow \text{term} \ast \text{factor} \mid \text{term} / \text{factor} \mid \text{factor} \]
\[ \text{expr} \rightarrow \text{expr} + \text{term} \mid \text{expr} - \text{term} \mid \text{term} \]

Example: \( \text{id} \ast \text{id} + \text{id} \uparrow \text{id} \uparrow \text{id} \)

Top-down Parsing

Idea: construct parse tree by starting at start symbol and “guessing” each derivation until we reach a string that matches input.

Example Grammar:
\[ S \rightarrow \text{if } E \text{ then } S \text{ else } S \mid \text{while } E \text{ do } S \mid \text{pause} \]
\[ E \rightarrow \text{true} \mid \text{false} \mid \text{id} \]

Token string:
\[ \text{if } \text{id} \text{ then while true do pause else pause} \]

Tree:

Input:
\[ \text{if } \text{id} \text{ then while true do pause else pause} \]

Action: Guess for \( S \)

Example:
\[ \text{id} \ast \text{id} + \text{id} \uparrow \text{id} \uparrow \text{id} \]

Input:
\[ \text{id} \ast \text{id} + \text{id} \uparrow \text{id} \uparrow \text{id} \]

Action:
\[ \text{if matches; guess for } E \]
Tree:

```
  S
  /\  /
 if E then S else S
  |  |
 id
```

Input: `id` then while true do pause else pause
Action: `id` matches; then matches; guess for `S`

---

Tree:

```
  S
  /\  /
 if E then S else S
  |  |
 id while E do S
```

Input: `true` do pause else pause
Action: `true` matches; do matches; guess for `S`

---

Tree:

```
  S
  /\  /
 if E then S else S
  |  |
 id while E do S
```

Input: while true do pause else pause
Action: while matches; guess for `E`

---

Tree:

```
  S
  /\  /
 if E then S else S
  |  |
 id while E do S
```

Input: pause else pause
Action: pause matches; else matches; guess for `S`
Tree:

```
S
  if E then S else S
  id while E do S
  pause
``` 

Input: pause

Action:

pause matches; input exhausted; done.

**Recursive-Descent Parsing**

Implementation of top-down parser using a recursive procedure for each non-terminal.

For many languages, can make perfect guesses (avoid back-tracking) by using 1-symbol lookahead. I.e., if

\[ A \rightarrow \alpha_1 | \alpha_2 | \cdots | \alpha_n \]

choose correct \( \alpha_i \) by looking at first symbol it derives.

(If \( \epsilon \) is an alternative, choose it last.)

This approach is also called **predictive parsing**.

R.D. parsers are easy to write by hand and reasonably efficient.

Often must massage grammar into suitable form (more later).

Not all languages can be parsed this way.

Grammars that can be parsed using predictive parsing with 1-symbol lookahead are said to be **LL(1)**. A language is **LL(1)** if it has an **LL(1)** grammar.

No ambiguous grammar can be **LL(1)**.

**Recursive-Descent Example**

\[
S \rightarrow \text{if } E \text{ then } S \text{ else } S | \text{while } E \text{ do } S | \epsilon
\]

\[
E \rightarrow \text{true } | \text{false } | \text{id}
\]

\[
s() \{
  \text{if } (\text{tok} == \text{IF}) \{
    \text{lex()}; /* set tok to next input token */
    \text{e()};
  \}
  \text{if } (\text{tok} == \text{THEN}) \{
    \text{lex()};
    \text{s()}; /* recursive call! */
    \text{if } (\text{tok} == \text{ELSE}) \{
      \text{lex()};
      \text{s()};
      \} \text{ else error(); /* issue error message */}
    \} \text{ else error();}
  \}
  \text{else if } (\text{tok} == \text{WHILE}) \{
    \text{lex()};
    \text{e()};
    \text{if } (\text{tok} == \text{DO}) \{
      \text{lex()};
      \text{s()};
      \} \text{ else error();}
    \}
  \} \text{ else error();}
\}
\]

\[
e() \{
  \text{if } (\text{tok} == \text{TRUE } | \text{tok} == \text{FALSE } | \text{tok} == \text{ID})
    \text{lex()};
  \text{else error();}
\}
\]

**Problems for Recursive-Descent Parsing**

- **Left recursion**: a derivation
  
  \[ A \Rightarrow A \alpha \]
  
  causes parser to loop!

  Solution: **Remove** left recursion from grammar.

- **Need to backtrack** (inefficient) because one-symbol lookahead can’t “guess” correctly, e.g.:

  \[
  S \rightarrow V := \text{int}
  \]

  \[
  V \rightarrow \text{alpha } [\text{int }] | \text{alpha}
  \]

  Possible inputs: \( x := 77 \) or \( x[2] := 17 \).

  Which alternative should we choose for \( V \)?

  Solution: **Left-factor** the grammar.

- These problems arise naturally in expression grammars. (Can usually prevent them in statement grammars by careful language design.)
Eliminating Immediate Left Recursion

Replace left-recursive productions of the form

\[ A \rightarrow A\alpha | \beta \]

which generate sentences of the form

\[ \beta, \beta\alpha, \beta\alpha\alpha, \ldots \]

by the right-recursive productions

\[ A \rightarrow \beta A' \]
\[ A' \rightarrow \alpha A' | \epsilon \]

Yields different parse trees but same language:

Example: Arithmetic Expressions

\[ T \rightarrow T \ast F | T \div F | F \]
becomes

\[ T \rightarrow F T' \]
\[ T' \rightarrow \ast F T' | \div F T' | \epsilon \]

Left-factoring

Easy to remove common prefixes by left-factoring, creating new non-terminals.

Change

\[ V \rightarrow \alpha \beta | \alpha \gamma \]

to

\[ V \rightarrow \alpha V' \]
\[ V' \rightarrow \beta | \gamma \]

Example:

\[ S \rightarrow V := \text{int} \]
\[ V \rightarrow \text{alpha ' [' int ']'} | \text{alpha} \]
becomes

\[ S \rightarrow V := \text{int} \]
\[ V \rightarrow \text{alpha } V' \]
\[ V' \rightarrow ' [' \text{int } '] | \epsilon \]