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 two modern languages: Java and Standard ML.
- Between 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 languages(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?

FORTRAN, COBOL, (Visual) BASIC, ALGOL-60, ALGOL-68, PL/I, C, C++, RPG, Pascal, Modula, Oberon, Lisp, Scheme, ML, Haskell, Ada, Prolog, Goedel, Snobol, ICON, …

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) {
    // return true if n has no
    // divisor in interval [2,n-1]
    for (int d = 2; d < n; d++)
        if (n % d == 0)
            return false;
    return true;
}
```

In Standard ML (using a recursive function):

```ml
fun isprime (n:int) : bool =
 (* return true if n has no
divisor in interval [2,n-1] *)
let
    fun no_divisor (d:int) : bool =
        (* return true if n has no
divisor in interval [d,n-1] *)
        (d >= n) orelse
        ((n mod d <> 0) andalso
         (no_divisor (d+1)))
    in
        no_divisor 2
    end
```
In Intel X86 assembler:

.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 ....
cld ; ... 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

\[
<program> ::= \text{BEGIN} <\text{statement-seq}> \text{END}
\]

\[
<\text{statement-seq}> ::= <\text{statement}>
\]

\[
<\text{statement-seq}> ::= <\text{statement}> ; <\text{statement-seq}>
\]

\[
<\text{statement}> ::= <\text{while-statement}>
\]

\[
<\text{statement}> ::= <\text{for-statement}>
\]

\[
<\text{statement}> ::= <\text{empty}>
\]

\[
<\text{while-statement}> ::= \text{WHILE} <\text{expression}> \text{DO} <\text{statement-seq}> \text{END}
\]

\[
<\text{expression}> ::= <\text{factor}>
\]

\[
<\text{expression}> ::= <\text{factor}> \text{AND} <\text{factor}>
\]

\[
<\text{expression}> ::= <\text{factor}> \text{OR} <\text{factor}>
\]

\[
<\text{factor}> ::= ( <\text{expression}> )
\]

\[
<\text{factor}> ::= <\text{variable}>
\]

\[
<\text{for-statement}> ::= \ldots
\]

\[
<\text{variable}> ::= \ldots
\]
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., \texttt{<program>}, \texttt{<expression>}, \texttt{<S>}

- Terminals are represented by themselves, e.g., \texttt{WHILE}, (, 3. The empty string is written as \texttt{<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

Program ::= BEGIN Statement-seq END

Statement-seq ::= Statement

[ '; ' Statement-seq ]

Statement ::= [ While-statement | For-statement ]

While-statement ::= WHILE Expression DO Statement-seq END

Expression ::= Factor { (AND | OR) Factor }

Factor ::= '( ' Expression ' ) ' | Variable

For-statement ::= . . .

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.

Note: Here we are using “language” in the technical sense meaning “a well-defined set of strings over the specified character set”.
Simple Example

Character set: \{ (, ) \}

Terminals: \{ (, ) \}

Productions:
S ::= (S)
S ::= SS
S ::= \epsilon \text{ (the empty string)}

Starting nonterminal: S

Sample derivation:
S \rightarrow (S) \rightarrow (SS) \rightarrow ((S)S) \rightarrow ((S)S) \rightarrow ((S)S) \rightarrow ((S)S) \rightarrow ((S)S) \rightarrow ((S)S) \rightarrow ((S)S) \rightarrow ((S)S)

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

It is often useful to think of a derivation as a tree (more shortly).
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.

- Parsers can be built by hand, but often are produced using a parser generator.

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.
Parse Trees

Graphical representation of a derivation.

Given this grammar:

\[
expr \rightarrow expr + expr \mid expr * expr \\
\quad \mid (expr) \mid -expr \mid id
\]

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

```
expr
  / \          
-   expr      
  / \          
(expr)     
  /   \        
expr + expr
    /   \     
  id x   id y
```

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

Leaves are labeled with terminals, which can have attributes (in this case the specific identifier name).

The derived sentence is found by reading leaves (or “fringe”) 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

$$a + b \times c$$

has two parse trees:

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.
Ambiguity in Arithmetic Expressions

A grammar such as

\[ E \rightarrow E + E \mid E - E \mid E \times E \mid E / E \]

\[ E \uparrow E \mid (E) \mid -E \mid \text{id} \]

is ambiguous about order of operations.

Want to define

- **Precedence** - which operation is done first?
- **Associativity** - is

\[ X \ op_1 \ Y \ op_2 \ Z \]

equivalent to \((X \ op_1 \ Y) \ op_2 \ Z\) or to \(X \ op_1 \ (Y \ op_2 \ Z)\) (assuming \(op_1\) and \(op_2\) have same precedence).

The “usual” rules (based on common usage in written math) give the following precedences, highest first:

- (unary minus)
- \((\text{exponentiation})\)
- \(* /\)
- \(+ -\)

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.

\[
\begin{align*}
atom & \rightarrow (expr) \mid id \\
primary & \rightarrow \neg primary \mid atom \\
factor & \rightarrow primary \uparrow factor \mid primary \\
term & \rightarrow term * factor \mid term / factor \mid factor \\
expr & \rightarrow expr + term \mid expr - term \mid term
\end{align*}
\]

Example: \( a * b + c \uparrow d \uparrow e \)
Limitations of Context-free Grammars

Context-free grammars are very useful for describing the structure of programming languages and identifying legal programs.

But there are many useful characteristics of legal programs that cannot be captured in a grammar (no matter how clever we are).

For example, in many programming languages, every variable in a legal program must be declared before it is used. But this property cannot be captured in a grammar.

To show this formally, we can abstract the notion of “declaration before use” into a formal language

\[ L = \{wcw \mid w \in (a \mid b)^*\} \]

It can be easily shown that no context-free grammar generates \( L \).

So checking legality of programs typically requires more than syntax analysis. Most compilers use a secondary “semantic” analysis phase to check non-syntactic properties, such as type-correctness. Of course, sometimes illegal programs cannot be detected until runtime.
Parse Trees vs. Abstract Syntax Trees

Parse trees reflect details of the **concrete** syntax of a program, which is typically designed for easy parsing.

For processing a language, we usually want a **simpler**, more **abstract** view of the program. (No firm rules about AST design: matter of taste, convenience.)

Simple concrete grammar:

\[
S \rightarrow \text{while } (' E ') \text{ do } S \mid ID '=' E \\
E \rightarrow E '+' T \mid E '-' T \mid T \\
T \rightarrow ID \mid NUM \mid '(' 'E ' ')
\]

Example concrete parse tree for

\[
\text{while } (n) \text{ do } n = n - (b + 1)
\]
Parse Trees vs. Abstract Syntax Trees
Possible abstract syntax tree for

```
while (n) do n = n - (b + 1)
```

![Abstract Syntax Tree Diagram]

AST’s obey a tree grammar. Rules have form

```
Label (attr_1) \ldots (attr_m) : Kind \rightarrow Kind_1 \ldots Kind_n
```

where the LHS classifies the possible node labels into kinds, and the RHS describes the kinds of children required.

```
While : Stmt \rightarrow Exp Stmt
Assgn : Stmt \rightarrow Exp
Add : Exp \rightarrow Exp Exp
Sub : Exp \rightarrow Exp Exp
Id (id) : Exp \rightarrow \epsilon
Num (int) : Exp \rightarrow \epsilon
```

(I’ve enhanced Scott’s definition of tree grammars to include attributes, such as the identifier carried by \texttt{Id} and the integer carried by \texttt{Num}.)
Internal Representation of ASTs in C

AST’s have recursive structure and irregular shape and size, so it makes sense to store them as heap data structures using one record for each tree node.

In C, Pascal, Ada, etc., we might use unions or variant records for the different node labels of each node kind. E.g., in C:

```c
struct stmt {
    enum { While, Assgn, ... } label;
    union {
        struct {
            struct exp *controller;
            struct stmt *body;
        } while_s;
        struct {
            char *lhs; struct exp *rhs;
        } assgn_s;
        ...
    } u;
}

struct exp {
    enum { Add, ..., Num } label;
    union {
        struct {
            struct exp *left; struct exp *right;
        } add_e;
        ...
        struct {
            int value;
        } num_e;
    } u;
}
```
AST’s in Java or ML

In Java, heap records are **objects**. We define **classes** corresponding to the various kinds and a subclass for each label, e.g.

```java
abstract class Stmt { }
class While extends Stmt {
    Exp controller;
    Stmt body;
}
class Assgn extends Stmt {
    String lhs; Exp rhs;
}
...
abstract class Exp { }
class Add extends Exp {
    Exp left; Exp right;
}
...
class Num extends Exp {
    int value;
}
```

In ML, we can use **datatypes** to define a suitable set of variants.

```ml
datatype stmt = While of exp * stmt
    | Assgn of string * exp
    | ...
and exp = Add of exp * exp
    | ...
    | Num of int
```

All these approaches generate roughly the **same** heap structures!
External Representation of ASTs

Although ASTs are designed as an **internal** program representation, it can be useful to give them an **external** form too that can be read or written by other programs or by humans.

Any external representation of ASTs must accurately reflect the internal tree structure as well as the “fringe” of the tree. Can’t use tree grammar to parse, since it is typically ambiguous!

One approach (deriving from the programming language LISP) is to use parenthesized prefix notation to represent trees.

Each node in the tree is represented by the expression

\[
( \text{label} \ \text{attr}_1 \ \ldots \ \text{attr}_n \ \text{child}_1 \ \text{child}_n )
\]

where \text{label} is the node label, the \text{attr}_i are the node’s attributes (if any), and the \text{child}_i are the node’s children (if any) – each of which is itself a node expression.

So the representation of our AST example would be

\[
(\text{While} \ (\text{Id} \ n) \\
\quad (\text{Assgn} \ n \ (\text{Sub} \ (\text{Id} \ n) \\
\quad \quad (\text{Add} \ (\text{Id} \ b) \\
\quad \quad \quad (\text{Num} \ 1)))))
\]

where the indentation is optional, but makes the representation easier for humans to read.

Parsing this representation is easy (especially if the numbers and kinds of attributes and children are fixed for a given node label).