Assignment 1

Any questions?

Overview of Java for C++ programmers

Java and C++: (approximately) C with objects
  Many similarities

Key differences
  Java is truly OO
  Java has automatic garbage collection

Everything in Java is in a class
No global variables
  Use static class variables
  Referred to as <classname>.<varname>

Even main is in a class
  Each class can redefine main
  Only parameter to main is list of strings

Running java applications:
  java <classname> <args>
Every class has a superclass except Object.
Object is default superclass.
Inheritance indicated with extends.
public class2 extends <superclass>
Only single inheritance.
Interfaces define abstract functions.
One class can implement multiple interfaces.

Classes grouped into packages.
Used for naming and visibility.
Packages form hierarchy separated by .
Package statement specifies package.
package <packagename>;
See classes in package by default.
Can also see package java.lang.
import statement adds more.
import <package-path>.<classname>;
import <package-path>*;

const in C++ is final in Java.
No enums in Java.
Use set of named final ints.
No bit fields.
No preprocessor.
Some data types have changed:

- boolean: true or false
- char: more than 8 bits (unicode)
- byte: 8 bit integer
- short: 16 bit integer
- int: 32 bit integer
- long: 64 bit integer
- float: 32 bit float
- double: 64 bit float

---

No explicit pointers
- Implicit pointers to all objects
- No pointer arithmetic

Zero pointer is called null

Garbage collection
- No explicit delete or free
- Memory automatically freed when no longer in use

---

Create new objects with new
- new <classname>(<args>)

Constructors much like C++
- <classname>(<args>) { ... }
missing operators
  .
  & to take address of
  * to dereference pointers
  sizeof
new operators
  instanceof yields class of operand
  >>> unsigned right shift
changed operators
  + works on strings to concatenate
  >> always does signed shift
  &&,|| work on both int and boolean

goto statement is gone (good riddance)
Only loops can be labeled
Continue and break now optionally take a label
  Can break out of or go to next iteration of an arbitrary enclosing loop

C and C++ require forward declarations
  must declare everything before it is used
Not in Java!
  Compiler will find class definition
  Compiler will scan forward within class
Need to specify location of other classes
  Define classpath appropriately
Many standard java libraries (packages)
Important for this course
  java.lang.*
  java.io.*
  java.util.*
Reasonable free doc with jdk man pages
Also documented by any book

Arrays are special kind of objects
Can indicate type array of class1 as:
  class1[]
Create new (empty) array of class1 as:
  new class1[]
length is special field of every array
Elements of arrays can be arrays
  Multi-dimensional arrays
  Sub-arrays can vary in length

String another special kind of object
Two interesting classes:
  String - immutable
  StringBuffer - can be modified
String constants yield Strings
Many functions defined for both
  length()
  charAt()
  substring()
  equals()
Java has strong support for threads
  “Simultaneous” execution in same memory space
synchronized keyword helps preserves consistency
synchronized method
  only one synchronized method can execute on same object at once
synchronized statement
  synchronized (exp) { … }

Exceptions similar to C++
  Only throw subclasses of java.lang.Throwable
throw statement throws exceptions
try […] catch (<exceptionclass> <var>) {…}
  Can have multiple catch clauses
  Can end with finally clause
finally clause always executed, no matter how try finishes
Methods must declare exceptions thrown
Must catch undeclared exceptions

Java has many GUI environments
  CodeWarrior, JBuilder, Forte, …
Command line environment from JDK
  Available for free from sun
  Installed on many machines
javadoc compile
  javac *.java
Everything depends on CLASSPATH environment variable
Indicates where classes are found

```
java <classpath> <args>
```
Runs main in class given by <classpath>
Passes <args> to it
Can give args java before classpath
Various options; man java to see list

Back to lexing

Automated tools to build lexers
lex, flex, jlex

You provide list of regular expressions
Plus code fragments

Produces complete lexer

Performance varies
lex generated lexers are somewhat slow
flex generators lexers are fast

jlex performance is unclear
Claimed to be good

We'll look at how it works later
Body of lex input is sequence of rules

Rules consist of 2 parts
   Regular expression
   Code fragment

Code fragment is any block enclosed in { }
   Must return value to generate token
   Can include other code

Some additional useful reg-exps

$ matches end of line
   foo$ matches foo at the end of the line

^ matches start of line

3 sections of input to jlex
   1st part copied to top of output .java file
   2nd part includes directives to jlex
   3rd part is regexp rules

   parts separated by %%

   same basic structure for lex and flex
Directives
   Write at start of line
   Begin with %

%line includes code to track the current line
%char includes code to track the current offset
%cup makes the lexer compatible with jcup
%eof{ <code> %eof}: <code> will be called at end of file time

For others see lex documentation

May need to define code to be inserted in lexical analyzer class
Enclose code in %{ <code> %}
   Each delimiter must be at start of line

<code> is copied into generated class

Can define additional functions
Can define fields

Some tokens are context based
   Inside quoted string
   Inside comments

Lexer states simplifies these cases
   Necessary for multiple line matching

States partition rules
   Only rules in current state considered
Lexer state described by name enclosed in `< >` before regexp

Can map rule to multiple states
  Use multiple comma-separated names

Need to declare states in directives
  `%state id1,id2,...`

default state YYINITIAL
  already declared

To change states
  Include call to `yybegin` in code fragment
  Argument to `yybegin` is state name

Call `yybegin` again to return to old state

consider standard C comments
  `/*` anything `*/`

use rules

```c
<YYINITIAL>"*/" { 
  yybegin(INCOMMENT); }
<INCOMMENT>"*/" { yybegin(YYINITIAL); }
```

CS 210-2  lexer states  28
Lexers are built as finite automata (FA)
   Edge for each letter in alphabet
   Some states are accepting
Fact: Regexps describe regular languages,
   finite automata (FA) recognize regular langs.

Consider these regexps:
   if
   [0-9]+   
   [a-z_,][a-z0-9_]*
   */[^\n]*\n
Implemented as this finite automata:
Trivial to implement FA in code; need 2D array (similar to graph rep.):
Each row represents one state
Each column represents one input char
State stored at (r, c) represents next state, given current state r and input char c
Nothing stored at (r, c) if no transition from r on input c

Example:

Longest match is important
How to implement?
Remember last accepting state
When no more valid edges
Return to previous accepting state
Error if no previous accepting
Lexer built from finite automata
How? We start with regexps…

Deterministic finite automata (DFA)
Assumed thus far
Easy to implement
Only 1 edge per label from any state;
transitions unambiguous

Non-deterministic finite automata (NFA)
Allows multiple edges with same label;
transition choices
Also has ε transitions

Fact: DFAs and NFAs have equivalent expressiveness

So, why bother with NFA?
Hard to implement

Because starting from regexps:
Hard to define DFA
Easy to define NFA
Building an NFA from a regexp:

Start with a single starting state
For each regexp production:
    Add tail-only portion of NFA
Has single edge, maybe some more stuff and a final state

primitive
  a
epsilon
alternative
  r1 | r2

concatenation
  r1 r2
repetition
  r1 *
5 rules can describe all reg-exprs

Keep building up bigger NFA

But want to implement DFA...

Must convert from NFA to DFA
  Assignment in 243
  Not responsible for it in this class

Grammars and regular expressions:
  Both define languages

Grammars are more powerful:
  Describe langs not described by regexp

\[ exp := id \mid exp + exp \mid exp - exp \mid ( exp ) \]

Regexp cannot describe matched ()
  No recursion in definitions

Grammars usually list of productions
  \[ exp \rightarrow id \]
  \[ exp \rightarrow exp + exp \]
  \[ exp \rightarrow exp - exp \]
  \[ exp \rightarrow ( exp ) \]

Left hand side is a non-terminal
Right hand side is sequence of terminals and non-terminals
Terminals are tokens from lexer
One non-terminal is specified as start
A sentence is any complete sequence of terminals in the language defined by the grammar

alternatively

A sentence has complete derivation from the start symbol

consider starting from \( \text{exp} \) for \( a+b-c \)

\[
\begin{align*}
\text{exp} & \\
\text{exp} + \text{exp} & \\
\text{id} + \text{exp} & \\
\text{id} + \text{exp} - \text{exp} & \\
\text{id} + \text{id} - \text{exp} & \\
\text{id} + \text{id} - \text{id} & \\
\end{align*}
\]

many possible derivations for sentences

Parse trees represent set of derivations

Root is always start symbol

Each derivation expand a node

Leaves are terminals
exp
exp + exp
id + exp
id + exp - exp
id + id - exp
id + id - id

exp
exp + exp
id + exp
id + exp - exp
id + id - exp
id + id - id

exp
exp + exp
id + exp
id + exp - exp
id + id - exp
id + id - id
Parsing finds a derivation of a particular sentence, given a grammar

Most of next 3 weeks:
Work on ways to construct parsers
Ways to build parse trees
We deal with realistic (non-trivial) grammars

Grammar may be ambiguous
Multiple trees for same sentence
\[ a + b - c \]

Many languages can be described:
By ambiguous grammar
Non-ambiguous grammar
True for (almost) all interesting grammars

Important points:
Ambiguous grammars describe language
Sentence still in or not in (decidable);
ambiguity wrt derivation only
Two common ways to fix grammars:

**Precedence**
- Provide ordering of productions
- Favor higher precedence rules

**Associativity**
- Can be left or right
- Prefer deriving rules from left or right

Can also rewrite grammar
- Introduce non-terminals to impose precedence

```
exp f exp + exp
exp f exp - exp
exp f id
exp f ( exp )
```

```
expfexp + term
exp f term
term f term - factor
term f factor
factor f id
factor f ( exp )
```