Java & Object-Oriented Programming

Java is a fairly new and very popular programming language designed to support secure, platform-independent programming.

It is a good alternative to C or C++, trading a bit of efficiency for easier programming, debugging and maintenance.

Java is routinely interpreted (at the byte-code level), making it significantly slower than compiled C or C++. However true Java compilers exist, and are becoming more widespread. (IBM’s Jalapeno project is a good example). When compiled, Java’s execution speed is close to that of C or C++.

Basic Notions

In Java data is either primitive or an object (an instance of some class).

All code is written inside classes, so Java programming consists of writing classes.

Primitive data types are quite close to those of C or C++:

<table>
<thead>
<tr>
<th>Java Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>(not a numeric type)</td>
</tr>
<tr>
<td>char</td>
<td>(Unicode, 16 bits)</td>
</tr>
<tr>
<td>byte</td>
<td></td>
</tr>
<tr>
<td>short</td>
<td></td>
</tr>
<tr>
<td>int</td>
<td></td>
</tr>
<tr>
<td>long</td>
<td>(64 bits)</td>
</tr>
<tr>
<td>float</td>
<td></td>
</tr>
<tr>
<td>double</td>
<td></td>
</tr>
</tbody>
</table>
Objects

- All Java objects are instances of classes.
- All objects are heap-allocated, with automatic garbage collection.
- A reference to an object, in a variable, parameter or another object, is actually a pointer to some object allocated within the heap.
- No explicit pointer manipulation operations (like * or -> or ++) are needed or allowed.

Example:

```java
class Point {
    int x, y;
    Point data = new Point();
}
```

- Declaring an object reference (like class Point) does not automatically allocate space for an object. The reference is initialized to null unless an explicit initializer is included.

- Fields are accessed just as they are in C:
  ```java
data.x references field x in object data.
```

- Object references are automatically checked for validity (null or non-null). Hence
  ```java
data.x = 0;
```
  forces a run-time exception if data contains null rather than a valid object reference.

- Java makes it impossible for an object reference to access an illegal address. A reference is either null or a pointer to a valid, type-correct object in the heap. (This makes Java programs far more secure and reliable than C or C++ programs).

Class Members

Classes contain members. Class members are either fields (data) or methods (functions).

Example:

```java
class Point {
    int x, y;
    void clear() {x=0; y=0;}
}
Point d = new Point();
d.clear();
```

A special method is a constructor.
A constructor has no result type. It is used only to define the initialization of an object after the object has been created.
Constructors may be overloaded.

class Point {
    int x, y;
    Point() {x=0; y=0;}
    Point(int xin, int yin) {
        x = xin; y = yin;
    }
}

Static Members

Class members may be static.
A static member is allocated only once—for all instances of the class.
Ordinary members (called instance members) apply only to a particular class instance (i.e., only one object created from the class definition).

class Point {
    int x, y;
    static int howMany = 0;
    Point() {x=0; y=0;
        howMany++;}
    static void reset() {
        howMany = 0;
    }
}

Static member functions (methods) may not access non-static data.
(Why?)
Static members are accessed using a class name rather than the name of an object reference.
For example,
    Point.reset();

Visibility of Class Members

Class members may be declared as public, private or protected.
Public members may be accessed from outside a class.
Private members of a class may be accessed only from with the class itself.
Protected members may be accessed only from with the class itself or from within one of its subclasses.
Members nor marked public, private or protected are shared at the package level—similar to C++’s friend mechanism.
Example:

```java
class Customer {
    int id;
    private int pinCode;
}
Customer me = new Customer();
me.id = 1234; //OK
me.pinCode = 7777;
//Compile-time error
```

In a class, a special method, `main`, declared as

```java
static public void main(String[] args)
```

is automatically executed when a class is run.

`main` is very useful as a “test driver” for auxiliary and library classes.

Final Members

A field may be declared `final` making it effectively a constant.

```java
class Point {
    int x,y;
    static final Point origin = new Point(0,0);
    Point(int xin, int yin) {
        x = xin; y = yin;
    }
}
```

Final fields may be used to create constants within a class:

```java
class Card {
    final static int Clubs = 1;
    final static int Diamonds = 2;
    final static int Hearts = 3;
    final static int Spades = 4;
    int suit = Spades;
}
```

Inside a class suit names are available for use without qualification. E.g.,

```java
int suit = Spades;
```

Outside a class, the field names must be qualified using the class name:

```java
Card c = new Card();
c.suit = Card.Clubs;
```

Methods may also be marked as `final`. This forbids redeclaration in a subclass, allowing a more efficient implementation. Security may also be improved if a key method is known to be unchangeable.

Java Arrays

In Java, arrays are implemented as a special kind of class. Arrays of primitive types are implemented as an object that contains a block of values within it. Arrays of objects are implemented as an object that contains a block of object references within it. Allocating an array of objects does not allocate the objects themselves. Hence within an array of objects, some positions may reference actual objects while other may contain `null` (this can be advantageous) if objects are large.

Multi-dimensional arrays are arrays of arrays. Arrays within an array need not all have the same size.

Hence we may see
int[][] TwoDim = new int[3][];
TwoDim[0] = new int[1];
TwoDim[1] = new int[2];
TwoDim[2] = new int[3];
The size of an array is part of its value; not its type.
Thus
int[] A = new int[10];
int[] B = new int[5];
A = B;
is valid.
Pascal showed that making an array's size part of its type is undesirable. (Why?)
Still, forcing an array to have a fixed size can be necessary (e.g., an array indexed by months). (How do we simulate a fixed-size array?).

Subclassing in Java

When a new class is defined in terms of an existing class, the new class extends the existing class. The new class inherits all public and protected members of its parent (or base) class.
The new class may add new methods or fields. It may also redefine inherited methods or fields.

```java
class Point {
    int x, y;
    Point(int xin, int yin) {
        x = xin; y = yin;
    }
    static float dist(Point P1, Point P2) {
        return (float) Math.sqrt((P1.x-P2.x)*(P1.x-P2.x) + (P1.y-P2.y)*(P1.y-P2.y));
    }
}
```

```java
class Point3 extends Point {
    int z;
    Point3(int xin, int yin, int zin) {
        super(xin,yin); z=zin;
    }
    static float dist(Point3 P1, Point3 P2) {
        float d=Point.dist(P1,P2);
        return (float) Math.sqrt((P1.z-P2.z)*(P1.z-P2.z) + d*d);
    }
}
```

Note that although Point3 redefines dist, the old definition of dist is still available by using the parent class as a qualifier (Point.dist).
The same is true for fields that are hidden when a field in a parent is redeclared.

Non-static methods are automatically virtual: a redefined method is automatically used in all inherited methods including those defined in parent classes that think they are using an earlier definition of the class.

Example:
```java
class C {
    void DoIt() {PrintIt();}
    void PrintIt() {
        println("C rules!");
    }
}
class D extends C {
    void PrintIt() {
        println("D rules!");
    }
    void TestIt() {DoIt();}
}
D dvar = new D();
dvar.TestIt();
D rules! is printed.
```
Static methods in Java are not virtual (this can make them easier to implement efficiently).

Abstract Classes and Methods

Sometimes a Java class is not meant to be used by itself because it is intentionally incomplete. Rather, the class is meant to be starting point for the creation (via subclassing) of more complete classes. Such classes are abstract.

Example:

```java
abstract class Shape {
    Point location;
}
class Circle extends Shape {
    float radius;
    float area() {
        return Math.pi*radius*radius;
    }
}
```

Methods can also be made abstract to indicate that their actual definition will appear in subclasses:

```java
abstract class Shape {
    Point location;
    abstract float area();
}
class Circle extends Shape {
    float radius;
    float area() {
        return Math.pi*radius*radius;
    }
}
```

Subtyping and Inheritance

We can use a subtyping mechanism, as found in C++ or Java, for two different purposes:

- We may wish to inherit the actual implementations of classes and members to use as the basis of a more complete or extended class. To inherit an implementation, we say a given class “extends” an existing class:

```java
class Derived extends Base {
    ...;
}
```

- Class Derived contains all of the members of Base plus any others it cares to add.
We may wish to inherit an interface—a set of method names and values that will be available for use. To inherit (or claim) an interface, we use a Java interface definition. An interface doesn't implement anything; rather, it gives a name to a set of operations or values that may be available within one or more classes.

Why are Interfaces Important?

Many classes, although very different, share a common subset of values or operations. We may be willing to use any such class as long as only interface values or operations are used.

For example, many objects can be ordered (or at least partially-ordered) using a “less than” operation.

If we always implement less than the same way, for example,

```java
boolean lessThan(Object o1, Object o2);
```

then we can create an interface that admits all classes that know about the lessThan function:

```java
interface Compare {
    boolean lessThan(Object o1, Object o2);
}
```

Now different classes can each implement the Compare interface, proclaiming to the world that they know how to compare objects of the class they define:

```java
class IntCompare implements Compare {
    public boolean lessThan(Object i1, Object i2) {
        return ((Integer)i1).intValue() < ((Integer)i2).intValue();
    }
}
class StringCompare implements Compare {
    public boolean lessThan(Object i1, Object i2) {
        return ((String)i1).compareTo((String)i2) < 0;
    }
}
```

The advantage of using interfaces is that we can now define a method or class that only depends on the given interface, and which will accept any type that implements that interface.

```java
class PrintCompare {
    public static void printAns(Object v1, Object v2, Compare c) {
        System.out.println(v1.toString() + " < " + v2.toString() + " is " + new Boolean(c.lessThan(v1, v2)).toString());
    }
}
class Test {
    public static void main(String args[]) {
        Integer i1 = new Integer(2);
        Integer i2 = new Integer(1);
        PrintCompare.printAns(i1, i2, new IntCompare());
        String s2 = "abcdef";
        String s1 = "xyzaa";
        PrintCompare.printAns(s1, s2, new StringCompare());
    }
}
```
Since classes may have many methods and modes of use or operation, a given class may implement many different interfaces. For example, many classes support the Clonable interface, which states that objects of the class may be duplicated (cloned).

**Exceptions in Java**

Java provides a fairly elaborate exception handling mechanism based on the throw-catch model.

All exceptions are objects, required to be a subclass of Throwable.

Class Throwable has two subclasses, Exception and Error. Class Exception has a subclass RuntimeException.

Exceptions may be explicitly thrown (using a throw statement) or they may be implicitly thrown as the result of a run-time error.

For example, an ArithmeticException is thrown for certain run-time arithmetic errors, like division by zero.

Unlike other languages, Java divides exceptions into two general classes: checked and unchecked.

A checked exception must either be caught (using a try-catch block) or propagated (by marking a method as throwing the exception).

This means that checked exceptions cannot be ignored—you must be prepared to catch them or you must “advertise” to your callers that you may throw an exception back to them.

Unchecked exceptions need not be caught or marked as potentially thrown. This makes exception handling for such exceptions optional. Unchecked exceptions are typically those that might occur so often (like NullPointerException or ArithmeticException) that forced checks could unnecessarily clutter a program without significant benefit.

How are checked and unchecked exceptions distinguished?

- Any exception that is a member (or subclass) of Error or RuntimeException is unchecked.
- All other exceptions must be checked.

Exceptions are propagated dynamically:

- When an exception is thrown (explicitly or implicitly) the innermost try-catch block that can “catch” the exception is selected, and
the catch block that matches the exception is executed.

- A catch block "catches" a given exception if the class of the exception is the same as the class used in the catch. An exception that is a subclass of the catch's exception class will also be caught. Thus an catch that handles class Throwable catches all exceptions.

- If no catch can handle the exception in the current method, a return to the method's caller is forced, and the exception is rethrown from the point of call.

- This process is repeated until a catch that can handle the exception is found or until we force a return from the main method.

- If a return from the main method is forced, no handler exists. A run-time error message is printed ("Uncaught exception") and execution is terminated.

- One of the limitations of Java's exception mechanism (and similar mechanisms found in other languages) is that there is no "retry" mechanism. Once an exception is thrown, we never go back to the point where the exception occurred. This is why Scheme's call/cc mechanism is considered so special and unique.

Example:

```java
class badValue extends Exception{
    float value;
    badValue(float f) {value=f;}
}

float sqrt(float val)
    throws badValue {
        if (val < 0.0)
            throw new badValue(val);
        else return (float) Math.sqrt(val);
    }

try {
    System.out.println("Ans = " + sqrt(-123.0));
} catch (badValue b) {
    System.out.println("Can't take sqrt of "+b);
}
```

Lex/Flex/JLex

Lex is a well-known Unix scanner generator. It builds a scanner, in C, from a set of regular expressions that define the tokens to be scanned.

Flex is a newer and faster version of Lex.

JLex is a Java version of Lex. It generates a scanner coded in Java, though its regular expression definitions are very close to those used by Lex and Flex.

Lex, Flex and JLex are largely non-procedural. You don't need to tell the tools how to scan. All you need to tell it what you want scanned (by giving it definitions of valid tokens).
This approach greatly simplifies building a scanner, since most of the details of scanning (I/O, buffering, character matching, etc.) are automatically handled.

---

JLex

JLex is coded in Java. To use it, you enter

```
java JLex.Main f.jlex
```

Your CLASSPATH should be set to search the directories where JLex's classes are stored. (The CLASSPATH we gave you includes JLex's classes).

After JLex runs (assuming there are no errors in your token specifications), the Java source file `f.jlex.java` is created. (`f` stands for any file name you choose. Thus `csx.jlex` might hold token definitions for CSX, and `csx.jlex.java` would hold the generated scanner).

---

You compile `f.jlex.java` just like any Java program, using your favorite Java compiler.

After compilation, the class file `Yylex.class` is created.

It contains the methods:

- Token `yylex()` which is the actual scanner. The constructor for `Yylex` takes the file you want scanned, so `new Yylex(System.in)` will build a scanner that reads from `System.in`. `Token` is the token class you want returned by the scanner; you can tell JLex what class you want returned.

- String `yytext()` returns the character text matched by the last call to `yylex`.

---

A simple example of using JLex is in `~cs536-1/public/jlex` just enter

```
make test
```
Input to JLex

There are three sections, delimited by `%%`. The general structure is:

**User Code**

**JLex Directives**

**Regular Expression rules**

The User Code section is Java source code to be copied into the generated Java source file. It contains utility classes or return type classes you need. Thus if you want to return a class `IntlitToken` (for integer literals that are scanned), you include its definition in the User Code section.

JLex directives are various instructions you can give JLex to customize the scanner you generate. These are detailed in the JLex manual. The most important are:

- `%{` Code copied into the Yylex class (extra fields or methods you may want) `}%`
- `%eof{` Java code to be executed when the end of file is reached `%eof}`
- `%type classname` classname is the return type you want for the scanner method, `yylex()`

Macro Definitions

In section two you may also define macros, that are used in section three. A macro allows you to give a name to a regular expression or character class. This allows you to reuse definitions and make regular expression rule more readable.

Macro definitions are of the form `name = def`

Macros are defined one per line. Here are some simple examples:

- `Digit=[0-9]`
- `AnyLet=[A-Za-z]`

In section 3, you use a macro by placing its name within `{ and }`. Thus `{Digit}` expands to the character class defining the digits 0 to 9.

Regular Expression Rules

The third section of the JLex input file is a series of token definition rules of the form

`RegExpr {Java code}`

When a token matching the given `RegExpr` is matched, the corresponding Java code (enclosed in `“{“ and “}”`) is executed. JLex figures out what `RegExpr` applies; you need only say what the token looks like (using `RegExpr`) and what you want done when the token is matched (this is usually to return some token object, perhaps with some processing of the token text).
Here are some examples:

```java
"+"   {return new Token(sym.Plus);};
(" ")+ {/* skip white space */}
(Digit)+ {return new
    IntToken(sym.Intlit,
    newInteger(yytext().intValue()));};
```

### Regular Expressions in JLex

To define a token in JLex, the user to associates a regular expression with commands coded in Java.

When input characters that match a regular expression are read, the corresponding Java code is executed. As a user of JLex you don’t need to tell it how to match tokens; you need only say what you want done when a particular token is matched.

Tokens like white space are deleted simply by having their associated command not return anything. Scanning continues until a command with a return in it is executed.

The simplest form of regular expression is a single string that matches exactly itself.

For example,

```java
if   {return new Token(sym.If);};
```

If you wish, you can quote the string representing the reserved word ("if"), but since the string contains no delimiters or operators, quoting it is unnecessary.

For a regular expression operator, like +, quoting is necessary:

```java
"+"   {return newToken(sym.Plus);}
```

### Character Classes

Our specification of the reserved word if, as shown earlier, is incomplete. We don’t (yet) handle upper or mixed-case.

To extend our definition, we’ll use a very useful feature of Lex and JLex—character classes.

Characters often naturally fall into classes, with all characters in a class treated identically in a token definition. In our definition of identifiers all letters form a class since any of them can be used to form an identifier. Similarly, in a number, any of the ten digit characters can be used.
Character classes are delimited by [ and ]; individual characters are listed without any quotation or separators. However, ^, ], and -, because of their special meaning in character classes, must be escaped. The character class [xyz] can match a single x, y, or z.

The character class [\]] can match a single ] or ). (The ] is escaped so that it isn't misinterpreted as the end of character class.)

Ranges of characters are separated by a –; [x–z] is the same as [xyz]. [0–9] is the set of all digits and [a–zA–Z] is the set of all letters, upper- and lower-case. \ is the escape character, used to represent unprintables and to escape special symbols.

Following C and Java conventions, \n is the newline (that is, end of line), \t is the tab character, \\ is the backslash symbol itself, and \010 is the character corresponding to octal 10.

The ^ symbol complements a character class (it is JLex's representation of the Not operation).

[^xy] is the character class that matches any single character except x and y. The ^ symbol applies to all characters that follow it in a character class definition, so [^0–9] is the set of all characters that aren't digits. ^ can be used to match all characters.

Here are some examples of character classes:

<table>
<thead>
<tr>
<th>Character Class</th>
<th>Set of Characters Denoted</th>
</tr>
</thead>
<tbody>
<tr>
<td>[abc]</td>
<td>Three characters: a, b and c</td>
</tr>
<tr>
<td>[cba]</td>
<td>Three characters: a, b and c</td>
</tr>
<tr>
<td>[a–c]</td>
<td>Three characters: a, b and c</td>
</tr>
<tr>
<td>[aabbcc]</td>
<td>Three characters: a, b and c</td>
</tr>
<tr>
<td>[^abc]</td>
<td>All characters except a, b and c</td>
</tr>
<tr>
<td>[^^-]]</td>
<td>Three characters: ^, – and ]</td>
</tr>
<tr>
<td>[^]</td>
<td>All characters</td>
</tr>
<tr>
<td>&quot;[abc]&quot;</td>
<td>Not a character class. This is one five character string: [abc]</td>
</tr>
</tbody>
</table>

**Regular Operators in JLex**

JLex provides the standard regular operators, plus some additions.

- Catenation is specified by the juxtaposition of two expressions; no explicit operator is used. Outside of character class brackets, individual letters and numbers match themselves; other characters should be quoted (to avoid misinterpretation as regular expression operators).

<table>
<thead>
<tr>
<th>Regular Expr</th>
<th>Characters Matched</th>
</tr>
</thead>
<tbody>
<tr>
<td>a b cd</td>
<td>Four characters: abcd</td>
</tr>
<tr>
<td>(a) (b) (cd)</td>
<td>Four characters: abcd</td>
</tr>
<tr>
<td>[ab] [cd]</td>
<td>Four different strings: ac or ad or bc or bd</td>
</tr>
<tr>
<td>while</td>
<td>Five characters: while</td>
</tr>
<tr>
<td>&quot;while&quot;</td>
<td>Five characters: while</td>
</tr>
<tr>
<td>[w] [h] [i] [l] [e]</td>
<td>Five characters: while</td>
</tr>
</tbody>
</table>

Case is significant.
The alternation operator is |. Parentheses can be used to control grouping of subexpressions. If we wish to match the reserved word while allowing any mixture of upper- and lowercase, we can use (w|W) (h|H) (i|I) (l|L) (e|E) or [wW][hH][iI][lL][eE]

- **Regular Expr** | **Characters Matched**
  - ab|cd | Two different strings: ab or cd
  - (ab)| (cd) | Two different strings: ab or cd
  - [ab]| [cd] | Four different strings: a or b or c or d

Postfix operators:
- * Kleene closure: 0 or more matches
  (ab)* matches λ or ab or abab or ababab ...
- + Positive closure: 1 or more matches
  (ab)+ matches ab or abab or ababab ...
- ? Optional inclusion:
  expr? matches expr zero times or once.
  expr? is equivalent to (expr) | λ and eliminates the need for an explicit λ symbol.
  [+-]?[0-9]+ defines an optionally signed integer literal.

Single match:
The character "." matches any single character (other than a newline).

Start of line:
The character ^ (when used outside a character class) matches the beginning of a line.

End of line:
The character $ matches the end of a line. Thus,

^A.*e$ matches an entire line that begins with A and ends with e.

Overlapping Definitions

Regular expressions map overlap (match the same input sequence).
In the case of overlap, two rules determine which regular expression is matched:
- The longest possible match is performed. JLex automatically buffers characters while deciding how many characters can be matched.
- If two expressions match exactly the same string, the earlier expression (in the JLex specification) is preferred. Reserved words, for example, are often special cases of the pattern used for identifiers. Their definitions are therefore placed before the
expression that defines an identifier token.

Often a “catch all” pattern is placed at the very end of the regular expression rules. It is used to catch characters that don’t match any of the earlier patterns and hence are probably erroneous. Recall that " . " matches any single character (other than a newline). It is useful in a catch-all pattern. However, avoid a pattern like . * which will consume all characters up to the next newline.

In JLex an unmatched character will cause a run-time error.

The operators and special symbols most commonly used in JLex are summarized below. Note that a symbol sometimes has one meaning in a regular expression and an entirely different meaning in a character class (i.e., within a pair of brackets). If you find JLex behaving unexpectedly, it’s a good idea to check this table to be sure of how the operators and symbols you’ve used behave. Ordinary letters and digits, and symbols not mentioned (like @) represent themselves. If you’re not sure if a character is special or not, you can always escape it or make it part of a quoted string.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning in Regular Expressions</th>
<th>Meaning in Character Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>Matches with ) to group sub-expressions</td>
<td>Represents itself.</td>
</tr>
<tr>
<td>)</td>
<td>Matches with ( to group sub-expressions</td>
<td>Represents itself.</td>
</tr>
<tr>
<td>[</td>
<td>Begins a character class</td>
<td>Represents itself.</td>
</tr>
<tr>
<td>]</td>
<td>Represents itself. Ends a character class</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td>Matches with } to signal macro-expansion</td>
<td>Represents itself.</td>
</tr>
<tr>
<td>}</td>
<td>Matches with [ to signal macro-expansion</td>
<td>Represents itself.</td>
</tr>
<tr>
<td>*</td>
<td>Matches with &quot; to delimit strings (only \ is special within strings)</td>
<td>Represents itself.</td>
</tr>
<tr>
<td>\</td>
<td>Escapes individual characters. Also used to specify a character by its octal code</td>
<td>Escapes individual characters. Also used to specify a character by its octal code.</td>
</tr>
<tr>
<td>.</td>
<td>Matches any one character except \n</td>
<td>Represents itself.</td>
</tr>
</tbody>
</table>
Potential Problems in Using JLex

The following differences from “standard” Lex notation appear in JLex:

- Escaped characters within quoted strings are not recognized. Hence "\n" is not a new line character. Escaped characters outside of quoted strings (\n) and escaped characters within character classes ([\n]) are OK.
- A blank should not be used within a character class (i.e., [ and ]). You may use \040 (which is the character code for a blank).

JLex Examples

A JLex scanner that looks for five letter words that begin with “P” and end with “T”.

This example is in ~cs536-1/public/jlex

The JLex specification file is:

class Token {
    String text;
    Token(String t){text = t;}
}
%
Digit=[0-9]
AnyLet=[A-Za-z]
Others=[0-9'&.]
WhiteSp=\040\n
// Tell JLex to have yylex() return a Token
%type Token

// Tell JLex what to return when eof of file is hit
%eofval{
    return new Token(null);
}%eofval
%
[\p\]AnyLet\]AnyLet\]AnyLet\]T\]\]WhiteSp+
{return new Token(yytext());}

((AnyLet))\]Others\]\]+{WhiteSp}+
{"*/skip*/"}
The Java program that uses the scanner is:

```java
import java.io.*;

class Main {
    public static void main(String args[]) throws java.io.IOException {
        Yylex lex = new Yylex(System.in);
        Token token = lex.yylex();

        while (token.text != null) {
            System.out.print("\t" + token.text);
            token = lex.yylex(); // get next token
        }
    }
}
```

In case you care, the words that are matched include:

- Pabst
- paint
- petit
- pilot
- pivot
- plant
- pleat
- point
- posit
- Pratt
- print

A JLex tester that looks for matches of regular expressions being tested. This example is in
~cs536-1/public/jlex.tester

The JLex specification file is:

```java
class Token {
    String text;
    Token(String t) { text = t; }
}
%
Digit=[0-9]
AnyLet=[A-Za-z]
Others=[0-9’&.]
WhiteSp=[\040\n]

// Tell JLex to have yylex() return a Token
%type Token
// Tell JLex what to return when eof of file is hit
%eofval{
    return new Token(null);
}
%

testRE {return new Token(yytext());}
(WhiteSp)+ {/*skip*/}
(.) {System.out.println("Illegal:" + yytext());}
```
The Java program that uses this scanner tester is:
import java.io.*;

class Main {
    public static void main(String args[])
    throws java.io.IOException {
        Yylex lex = new Yylex(System.in);
        Token token = lex.yylex();

        while ( token.text != null ) {
            System.out.print("Matched:" +
            token.text);
            token = lex.yylex(); // get next token
        }
    }
}

An example of CSX token specifications. This example is in
~cs536-1/public/proj2/startup

The JLex specification file is:
import java_cup.runtime.*;

    /* Expand this into your solution for
    project 2 */

    class CSXToken {
        int linenum;
        int colnum;
        CSXToken(int line, int col){
            linenum=line; colnum=col;};
    }

    class CSXIntLitToken extends CSXToken {
        int intValue;
        CSXIntLitToken(int val, int line, int col){
            super(line,col); intValue=val;};
    }

    class CSXIdentifierToken extends CSXToken {
        String identifierText;
        CSXIdentifierToken(String text, int line, int col){
            super(line,col); identifierText=text;};
    }

    // This class is used to track line and
    // column numbers
    // Feel free to change to extend it
    class Pos {
        static int linenum = 1;
        /* maintain this as line number current
        token was scanned on */
        static int colnum = 1;
        /* maintain this as column number
        current token began at */
        static int line = 1;
        /* maintain this as line number after
        scanning current token */
The Java program that uses this scanner (P2) is:

class P2 {
    public static void main(String args[]) throws java.io.IOException {
        if (args.length != 1) {
            System.out.println("Error: Input file must be named on command line.");
            System.exit(-1);
        }
        java.io.FileInputStream yyin = null;
        try {
            yyin = new java.io.FileInputStream(args[0]);
        } catch (FileNotFoundException found) {
            System.out.println("Error: unable to open input file.");
            System.exit(-1);
        }
        // lex is a JLex-generated scanner that will read from yyin
        Yylex lex = new Yylex(yyin);

        System.out.println("Begin test of CSX scanner.");

        /***********************
        You should enter code here that thoroughly test your scanner.
        Be sure to test extreme cases, like very long symbols or lines, illegal tokens, unrepresentable integers, illegals strings, etc.
        The following is only a starting point.
        ***********************/
        Symbol token = lex.yylex();
        while (token.sym != sym.EOF) {
            System.out.print(((CSXToken) token.value).linenum + " ");
            System.out.println(((CSXToken) token.value).colnum + " ");
            switch (token.sym) {
                case sym.INTLIT:
                    System.out.println("integer literal(" + ((CSXIntLitToken) token.value).intValue + ")");
                    break;
            }
        }
    }
}
Java CUP

Java CUP is a parser-generation tool, similar to Yacc.

CUP builds a Java parser for LALR(1) grammars from production rules and associated Java code fragments.

When a particular production is recognized, its associated code fragment is executed (typically to build an AST).

CUP generates a Java source file parser.java. It contains a class parser, with a method
Symbol parse()
The Symbol returned by the parser is associated with the grammar’s start symbol and contains the AST for the whole source program.

Java CUP Specifications

Java CUP specifications are of the form:
- Package and import specifications
- User code additions
- Terminal and non-terminal declarations
- A context-free grammar, augmented with Java code fragments

Package and Import Specifications

You define a package name as:
package name ;
You add imports to be used as:
import java_cup.runtime.*;

The file sym.java is also built for use with a JLex-built scanner (so that both scanner and parser use the same token codes).

If an unrecovered syntax error occurs, Exception() is thrown by the parser.

CUP and Yacc accept exactly the same class of grammars—all LL(1) grammars, plus many useful non-LL(1) grammars.

CUP is called as
java java_cup.Main < file.cup
User Code Additions
You may define Java code to be included within the generated parser:

action code { /*java code */ :}
This code is placed within the generated action class (which holds user-specified production actions).

parser code { /*java code */ :}
This code is placed within the generated parser class.

init with{ /*java code */ :}
This code is used to initialize the generated parser.

scan with{ /*java code */ :}
This code is used to tell the generated parser how to get tokens from the scanner.

Terminal and Non-terminal Declarations
You define terminal symbols you will use as:

terminal classname name_1, name_2, ...
classname is a class used by the scanner for tokens (CSXToken, CSXIdentifierToken, etc.)

You define non-terminal symbols you will use as:

non terminal classname name_1, name_2, ...
classname is the class for the AST node associated with the non-terminal (stmtNode, exprNode, etc.)

Production Rules
Production rules are of the form
name ::= name_1 name_2 ... action ;
or
name ::= name_1 name_2 ... action_1
| name_3 name_4 ... action_2
| ...
Names are the names of terminals or non-terminals, as declared earlier.
Actions are Java code fragments, of the form
{ /*java code */ :}
The Java object associated with a symbol (a token or AST node) may be named by adding a :id suffix to a terminal or non-terminal in a rule.
RESULT names the left-hand side non-terminal.

The Java classes of the symbols are defined in the terminal and non-terminal declaration sections.
For example,

prog ::= LBRACE:l stmts:s RBRACE
{| RESULT=
new csxLiteNode(s,
l.linenum, l.colnum); :}

This corresponds to the production
prog → { stmts }
The left brace is given the name l; the stmts non-terminal is called s.
In the action code, a new CSXLiteNode is created and assigned to prog. It is constructed from the AST node associated with s. Its line and column numbers are those given to the left brace, l (by the scanner).
To tell CUP what non-terminal to use as the start symbol (prog in our example), we use the directive:
start with prog;

Example
Let’s look at the CUP specification for CSX-lite. Recall its CFG is

program → { stmts }
stmts → stmt stmts
| λ
stmt → id = expr ;
| if ( expr ) stmt
expr → expr + id
| expr - id
| id

The corresponding CUP specification is:

```java
/**
 * This Is A Java CUP Specification For
 * CSX-lite, a Small Subset
 * of The CSX Language, Used In Cs536
 */

/* Preliminaries to set up and use
the scanner. */
import java_cup.runtime.*;
parser code {:
    public void syntax_error
    (Symbol cur_token){
        report_error(
            "CSX syntax error at line "+
            String.valueOf(((CSXToken)
            cur_token.value).linenum),
            null);}
};

init with {:              :};
scan with {:
    return Scanner.next_token();
};
```

/* Terminals (tokens returned by the scanner). */
terminal CSXIdentifierToken IDENTIFIER;
terminal CSXToken SEMI, LPAREN,
RPAREN, ASG, LBRACE, RBRACE;
terminal CSXToken PLUS, MINUS,
rw_IF;

/* Non terminals */
non terminal csxLiteNode prog;
non terminal stmtsNode stmts;
non terminal stmtNode stmt;
non terminal exprNode expr;
non terminal nameNode ident;

start with prog;
prog::= LBRACE:l stmts:s RBRACE
{: RESULT=
    new csxLiteNode(s,
        l.linenum,l.colnum); :}
;
stmts::= stmt:s1 stmts:s2
{: RESULT=
  new stmtsNode(s1,s2,
  s1.linenum,s1.colnum);
:}
|
{: RESULT= stmtsNode.NULL; :}

stmt::= ident:id ASG exp:e SEMI
{: RESULT=
  new asgNode(id,e,
  id.linenum,id.colnum);
:}
|
rw_IF:i LPAREN exp:e RPAREN stmt:s
{: RESULT=new ifThenNode(e,s,
  stmtNode.NULL,
  i.linenum,i.colnum); :}

exp::=
exp:leftval PLUS:op ident:rightval
{: RESULT=new binaryOpNode(leftval,
  sym.PLUS, rightval,
  op.linenum,op.colnum); :}
|
exp:leftval MINUS:op ident:rightval
{: RESULT=new binaryOpNode(leftval,
  sym.MINUS, rightval,
  op.linenum,op.colnum); :}
|
ident::= IDENTIFIER:i
{: RESULT = i; :}

Let's parse
{ a = b ; }
First, a is parsed using
ident::= IDENTIFIER:i
{: RESULT = new nameNode(
  new identNode(i.identifierText,
  i.linenum,i.colnum),
  exprNode.NULL,
  i.linenum,i.colnum); :}

We build

Next, a is parsed using
ident::= IDENTIFIER:i
{: RESULT = new nameNode(
  new identNode(i.identifierText,
  i.linenum,i.colnum),
  exprNode.NULL,
  i.linenum,i.colnum); :}

We build
Then b's subtree is recognized as an exp:

\[
\text{ident}:i
\]
{ RESULT = i; }

Now the assignment statement is recognized:

\[
\text{stmt}::= \text{ident}:id \text{ ASG } \text{exp}:e \text{ SEMI}
\]
{ RESULT=
  \text{new } \text{asgNode}(\text{id}, e, \\
  \text{id}.\text{linenum}, \text{id}.\text{colnum});

We build

The stmts \(\rightarrow\lambda\) production is matched (indicating that there are no more statements in the program).

CUP matches

\[
\text{stmts}::= \{ \text{RESULT=} \text{stmtsNode}.\text{NULL}; \}
\]
and we build

Next,

\[
\text{stmts} \rightarrow \text{stmt} \text{ stmts}
\]
is matched using

\[
\text{stmts}::= \text{stmt}:s1 \text{ stmts}:s2
\]
{ RESULT=
  \text{new } \text{stmtsNode}(s1, s2, \\
  s1.\text{linenum}, s1.\text{colnum});

This builds

As the last step of the parse, the parser matches

\[
\text{program} \rightarrow \{ \text{stmts} \}
\]
using the CUP rule

\[
\text{prog}::= \text{LBRACE}:l \text{ stmts}:s \text{ RBRACE}
\]
{ RESULT=
  \text{new } \text{csxLiteNode}(s, \\
  l.\text{linenum}, l.\text{colnum}); }

The final AST returned by the parser is
Sample Type-Checking Routines for CSX-Lite

// abstract superclass; only subclasses are actually created
abstract class ASTNode {
    // Total number of type errors found
    static int typeErrors = 0;

    static void typeMustBe(int testType, int requiredType, String errorMsg) {
        if ((testType != Types.Error) &&
            (testType != requiredType)) {
            System.out.println(errorMsg);
            typeErrors++;
        }
    }

    String error() {
        return "Error (line " + linenum + ")";
    }
}

public static SymbolTable st =
    new SymbolTable();

void checkTypes();

// This node is used to root only
// CSX lite programs
class csxLiteNode extends ASTNode {
    void checkTypes()
    {
        fields.checkTypes();
        progStmts.checkTypes();
    }

    boolean isTypeCorrect() {
        checkTypes();
        return (typeErrors == 0);
    }

    private stmtsNode progStmts;
    private fieldDeclsNode fields;
}

class classNode extends ASTNode {
    // You need to refine this one
    boolean isTypeCorrect() {
        return true;
    }

    private identNode className;
    private memberDeclsNode members;
}

class fieldDeclsNode extends ASTNode {
    void checkTypes() {
        thisField.checkTypes();
        moreFields.checkTypes();
    }

    private declNode thisField;
    private fieldDeclsNode moreFields;
}

static void typesMustBeEqual(
    int type1, int type2, String errorMsg)
{
    if ((type1 != Types.Error) &&
        (type2 != Types.Error) &&
        (type1 != type2)) {
        System.out.println(errorMsg);
        typeErrors++;
    }
}

String error() {
    return "Error (line " + linenum + ")";
}

void checkTypes();

// This will normally need to be
// redefined in a subclass

// Root of all ASTs for CSX
class classNode extends ASTNode {
    // You need to refine this one
    boolean isTypeCorrect() {
        return true;
    }

    private identNode className;
    private memberDeclsNode members;
}

class fieldDeclsNode extends ASTNode {
    void checkTypes() {
        thisField.checkTypes();
        moreFields.checkTypes();
    }

    private declNode thisField;
    private fieldDeclsNode moreFields;
}
class nullFieldDeclsNode extends fieldDeclsNode {
    void checkTypes(){};
};

class varDeclNode extends declNode {
    void checkTypes() {
        SymbolInfo id;
        id = (SymbolInfo) st.localLookup(varName.idname);
        if (id != null) {
            System.out.println(error() + id.name() + " is already declared.");
            typeErrors++;
            varName.type = new Types(Types.Error);
        } else {
            id = new SymbolInfo(varName.idname, new Kinds(Kinds.Var), varType.type);
            varName.type = varType.type;
            try {
                st.insert(id);
            } catch (DuplicateException d) {
                /* can’t happen */
            } catch (EmptySTException e) {
                /* can’t happen */
                varName.idinfo=id;
            }
        }
    }
};

private identNode varName;
private typeNode varType;
private exprNode initValue;

abstract class typeNode extends ASTNode {
    // abstract superclass; only subclasses are actually created
    Types   type;
    // Used for typechecking
    // -- the type of this typeNode
};

class intTypeNode extends typeNode {
    intTypeNode(int line, int col) {
        super(line, col, new Types(Types.Integer));
    }
    void checkTypes() {
        // No type checking needed
    }
};

class stmtsNode extends ASTNode {
    void checkTypes() {
        thisStmt.checkTypes();
        moreStmts.checkTypes();
    }
    private stmtNode thisStmt;
    private stmtsNode moreStmts;
};

class nullStmtsNode extends stmtsNode {
    void checkTypes(){};
};

class asgNode extends stmtNode {
    void checkTypes() {
        target.checkTypes();
        source.checkTypes();
        // In CSX-lite all IDs are vars!
        assert(target.kind.val == Kinds.Var);
        typesMustBeEqual(source.type.val, target.type.val,
                         error() + "Both the left and right" +
                         " hand sides of an assignment must " +
                         "have the same type.");
    }
    private nameNode target;
    private exprNode source;
};
class ifThenNode extends stmtNode {
    void checkTypes() {
        condition.checkTypes();
        typeMustBe(condition.type.val, Types.Boolean,
            error() +
            "The control expression of an" +
            " if must be a bool.");
        thenPart.checkTypes();
        // No else parts in CSX Lite
    }
    private exprNode condition;
    private stmtNode thenPart;
    private stmtNode elsePart;
};

class printNode extends stmtNode {
    void checkTypes() {
        outputValue.checkTypes();
        typeMustBe(outputValue.type.val, Types.Integer,
            error() +
            "Only int values may be printed.");
    }
    private exprNode outputValue;
    private printNode morePrints;
};

// abstract superclass;
// only subclasses are actually
//created
abstract class exprNode extends ASTNode {
    protected Types   type;
    // Used for typechecking:
    // the type of this node
    protected Kinds   kind;
    // Used for typechecking:
    // the kind of this node
};

class binaryOpNode extends exprNode {
    void checkTypes() {
        //Only two bin ops in CSX-lite
        assert(operatorCode== sym.PLUS
            || operatorCode== sym.MINUS);
        leftOperand.checkTypes();
        rightOperand.checkTypes();
        type = new Types(Types.Integer);
        typeMustBe(leftOperand.type.val, Types.Integer,
            error() +
            "Left operand of" +
            toString(operatorCode) +
            "must be an int.");
        typeMustBe(rightOperand.type.val, Types.Integer,
            error() +
            "Right operand of" +
            toString(operatorCode) +
            "must be an int.");
    }
    private exprNode leftOperand;
    private exprNode rightOperand;
    private int operatorCode;
};

class identNode extends exprNode {
    void checkTypes() {
        SymbolInfo    id;
        //In CSX-lite all IDs are vars!
        assert(kind.val == Kinds.Var);
        id = (SymbolInfo)
            st.localLookup(idname);
        if (id == null) {
            System.out.println(error() +
                idname + " is not declared.");
            typeErrors++;
            type = new Types(Types.Error);
        } else {
            type = id.type;
            idinfo = id;
            // Save ptr to sym table entry
        }
    }
    publicString idname;
    public SymbolInfo idinfo;
    // sym table entry for this ident
    private boolean nullFlag;
};
class intLitNode extends exprNode {
    void checkTypes() {
        // All intLits are automatically type-correct
        private int intval;
    }
}

class nameNode extends exprNode {
    void checkTypes() {
        varName.checkTypes();
        // Subscripts not in CSX Lite
        type=varName.type;
    }
    private identNode varName;
    private exprNode subscriptVal;
}