CSc 413: Software Development

Dr. Barry Levine
TH 967
levine@sfsu.edu

SYLLABUS for CSc 413 - *Software Development*

Texts:

Understanding Object-Oriented Programming with Java, Budd, T., Addison-Wesley, 2000
Core Java(TM) 2, Volume I--Fundamentals, Horstmann, C.S. and Cornell, G. Prentice-Hall

Topics:

Introduction to Software Development

Introduction to Object-Oriented Programming - OOP
Information Hiding, Class Hierarchy

The Java Language

Chapter 3 - Object Oriented Design
Component, Responsibility, Collaborator Cards (CRC)
Plan for Change, Software Components, Interfaces vs. Implementation
Naming

A Comparision of Java and C++

A Compiler
Extended Example, Source, Tokens, AST, Decorated AST's, Code generation:
Bytecodes:

Lexical Analysis

Parsing - Syntax Analysis of the Token Stream Yielding the AST
Grammar for X, ASTS Built from Source Programs

Tree Visitors

Chapter 8 - Inheritance
Subclass, Subtypes and Substitutability, Forms of Inheritance, Modifiers
Benefits of Inheritance, Cost of Inheritance

The Interpreter
Frames (Activation Records)
Javadoc Documentation of Selected Interpreter Classes
The Runtime Stack, The Virtual Machine

Constraining (Decorating the AST; Type Checking)
Variable Scopes, Symbol Tables, Constraining Activities:

Code Generation
Frames (Activation Records),Runtime stack, Blocks

Chapter 10 - Mechanisms for Software Reuse
Inheritance vs. Composition (aggregation), Abstract classes vs. Interfaces, Combining Composition and Inheritance, Dynamic Composition

Chapter 11 - Implications of Inheritance
Polymorphic Variables, Memory Layout, Assignment, Clones (Shallow vs. Deep) Garbage Collection

Chapter 12 - Polymorphism
Polymorphic Variables, Overloading, Overriding, Replacement and Refinement
Abstract Methods, Efficiency and Polymorphism

Chapter 14 - Input and Output Streams - Effective Uses of Inheritance with Composition
Readers, InputStreams

Chapter 16 - Exception Handling in Java

Chapter 19 - Collection Classes
Arrays, Lists, Properties, System Properties

Application Profiling
Used to tune performance

PROGRAMS:

You will use the Java language for program development. Always use my schemas, ids, data structures and procedures where I specify them. **PROGRAMS THAT ARE NOT WORKING CORRECTLY WILL RECEIVE MINIMAL PARTIAL CREDIT.** Pretty print (indentation, etc.) and comment programs appropriately. Grades will be based on correctness, readability, the appropriate applications of object oriented techniques, efficiency, etc. ***IT IS A REQUIREMENT YOU WORK INDEPENDENTLY ON YOUR PROGRAMS IN THIS COURSE***.

ilearn Course Management System

We will use the ilearn course management system during this term. If you have used touch-tone registration for the course then you are already registered in this system. Use your SFSU id and personal access code to login to the system. The address to use for login is: ***https://ilearn.sfsu.edu/login/index.php.*** **Be sure to update your user profile to include the email address where you wish to receive course email from me**.

If you don't have an SFSU email account then you should first browse to http://www.sfsu.edu/infotech.htm and follow the *Help Desk* link to determine how to obtain an SFSU internet account.

Be sure to browse the ilearn system as soon as possible to learn about the resources available therein – e.g. discussion lists/FAQ's, grades, announcements.
I will frequently make announcements (change due dates, etc.) which will not appear in print. YOU are responsible for all announcements made in class!! I will also post messages to the ilearn system.

You are encouraged to interact in ilearn discussions. You may send private email messages to me. However, if your messages do not divulge programming solutions then I will require you to use ilearn. Subsequently, I would like class members to provide answers. This style of interaction is used extensively in practice so it's important for you to gain relevant experience. Furthermore, it will help you earn the 4 points for participation, etc.

Late Assignments:

Late programs will be downgraded 20% per day, based on the time the email submission is made.

Software:

You are expected to choose Netbeans with the latest version of JDK. Information on downloading/installing the free *netbeans* IDE is found at http://www.oracle.com/technetwork/java/index.html. **If you do not own your own pc/mac then you are welcome to use campus facilities - the departmental Science lab or other labs around campus (e.g. the Math Lab on 4th floor Thornton or the Business Lab)**

Attendance:

"Students are expected to attend classes regularly because classroom work is one of the necessary and important means of learning and of attaining the educational objectives of the institution." (SFSU Bulletin) To this end, attendance will be taken at different times through the term. Students missing class on a day of attendance will lose attendance points.

GRADING : (approximate)

PROGRAMS	35 %	
MIDTERM	29 %	
FINAL	29 %	
ATTENDANCE	3 %	
MISC	4 %	(e.g participation in lectures, ilearn, etc.)

Missed Exams:

Generally, there will be no make-up exams and no incomplete grades given. If you miss an exam, you must notify me before the exam or, if physically impossible, soon after. If any of the scheduled exam dates are in conflict with your religious observances, you must notify me, in writing, at least two weeks in advance of the exam. If you have an acceptable, documented excuse, you may be given a make-up exam or be given the average score of other exam(s) at the discretion of the instructor. Note that a make-up exam will consist of questions/exercises that might have a degree of difficulty that does not match those on the original exam.

Course Objectives and Role in Program

The objectives of this course include:

- Teach important object oriented programming principles using a large application as a vehicle for learning; consider issues of *programming in the large*
- Introduce the student to integrated development environments.
- Teach the Java programming language, along with well-utilized Java library resources
- Expose the student to other software development tools including debuggers and code profilers

Students will develop several small applications and at least one large application.
The knowledge of software development tools and object oriented programming plays an important role in all software development projects students develop for courses in the program.

Learning Outcomes

At the end of this course students will

- Be able to write Java programs utilizing an integrated development environment
- Utilize a debugger when doing software development
- Apply object oriented programming principles effectively when developing small to medium sized projects

- Write robust code utilizing exception handling language features
- Use a code profiler to tune a program's performance

Course Practices from Prior Terms:

Any rules and associated practices announced from past terms are not relevant for this term. This syllabus reflects the rules that will be applied this term.

Disability Access

Students with disabilities who need reasonable accommodations are encouraged to contact the instructor. The Disability Programs and Resource Center (DPRC) is available to facilitate the reasonable accommodations process. The DPRC is located in the Student Services Building and can be reached by telephone (voice/TTY 415-338-2472) or by email (dprc@sfsu.edu). http://www.sfsu.edu/~dprc

Student Disclosures of Sexual Violence

SF State fosters a campus free of sexual violence including sexual harassment, domestic violence, dating violence, stalking, and/or any form of sex or gender discrimination. If you disclose a personal experience as an SF State student, the course instructor is required to notify the Dean of Students. To disclose any such violence confidentially, contact:
The SAFE Place - (415) 338-2208; http://www.sfsu.edu/~safe_plc/
Counseling and Psychological Services Center - (415) 338-2208; http://psyservs.sfsu.edu/

For more information on your rights and available resources: http://titleix.sfsu.edu

Policy on Observance of Religious Holidays (F00-212)

The faculty of San Francisco State University shall make reasonable accommodations for students to observe religious holidays when such observances require students to be absent from class activities. It is the responsibility of the student to inform the instructor, in writing, about such holidays during the first two weeks of the class each semester. If such holidays occur during the first two weeks of the semester, the student must notify the instructor, in writing, at least three days before the date that he/she will be absent. It is the responsibility of the instructor to make every reasonable effort to honor the student request without penalty, and of the student to make up the work missed.

Computer Science Department Policy on Academic Cheating and Plagiarism

Official policy: http://cs.sfsu.edu/plagarism.html
An announcement of the existence of the policy will be made by the instructor at the first meeting each semester of every Computer Science class, at which time any course specific guidelines on cooperation and use of published programs can also be stated. For lower division courses, reference to this policy and statement of any course specific guidelines will be included in the course syllabus distributed to all students.

I. Introduction to Software Development

What will YOU encounter when you start programming

Programming-in-the-large vs. *Programming-in-the-small*

- Large programs/systems are hard to understand and very complex
- How do we organize our thoughts to understand, modify these systems?
- How do we organize programming teams to develop systems?
- What are the principles used to organize and manage large systems?
- How do we analyze and design these systems? OOA/OOD

You will most likely study code that has been developed so you can modify it - this might be overwhelming

- Be patient
- Be confident that you can do it
- Be aware of the organizational cultures within which you operate!

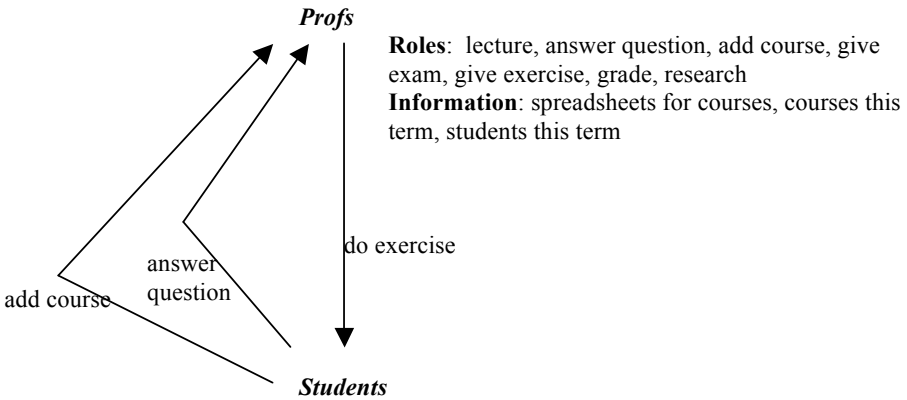
What tools do you use to study the code?

- Debuggers and documentation and communication with co-workers
- Texts from school on e.g. data structures, network programming, operating systems,
- The Web
- Manuals - dry, sleep-inducing
- Understand coding conventions and principles in your organization (and others since you're studying code developed by many organizations)
- Code profilers to study the efficiency of the components

How will CS 413 help prepare you for your programming job?

- You will study the foundation principles of OOP that are used to produce flexible, robust, re-usable, modifiable, quality software
- You will study Java - an exciting, powerful language and ENVIRONMENT that provides the mechanism for delivering state-of-the-art applications in a distributed, heterogeneous (web-based) environment
- You will study a compiler for a block-structured language written in Java
- It's a large program with code that illustrates the use of the important OOP principles covered in the course
- It's a large program so you will be confronted with the task of understanding the system
- You will modify components of the compiler
    1. Fix bugs
    2. Extend the system
- You will develop a component using specifications provided by the instructor
- You will be given the opportunity to be creative in your extensions.

II. Introduction to Object-Oriented Programming - OOP



Information Hiding

Prof keeps spreadsheets private

Student can turn in an exercise, prof will grade it and enter in spreadsheet (student isn't concerned with HOW prof keeps spreadsheet, just that prof grades and records)

Prof doesn't get involved in HOW student maintains course notes, just that student does it

FOCUS on ***WHAT*** the roles are (and mean) but ***NOT HOW*** they're implemented - helps *deal with complexity*

Instances of Profs:

Dr. Levine	Dr. Eisman
lecture	lecture
answer question	answer question
….	….
_____	_____
spreadsheet	spreadsheet
courses this term	courses this term
students this term	students this term

Instances of Students

Jane	Armen	Helen
study	study	study
do exercise	do exercise	do exercise
…	…	….
----------	----------	----------
courses completed	courses completed	courses completed
courses this term	courses this term	courses this term

Jane (Armen, Helen) can send messages to Levine/Eisman -- Levine and Eisman are ***receivers*** of ***messages***

Levine/Eisman have ***methods*** that will take the information in the messages and ***perform a computation*** based on this information. The computation is performed in the body of the function - **METHOD** - corresponding to the message

Messages: consist of a ***receiver and associated information*** used to process the message; the method processes the message

Message Originator ➔ ***Client***

Message Receiver ➔ ***Server*** (provides *service* requested by client)

Profs describe roles/behavior/services/responsibilities provided by the entire ***CLASS*** of Profs

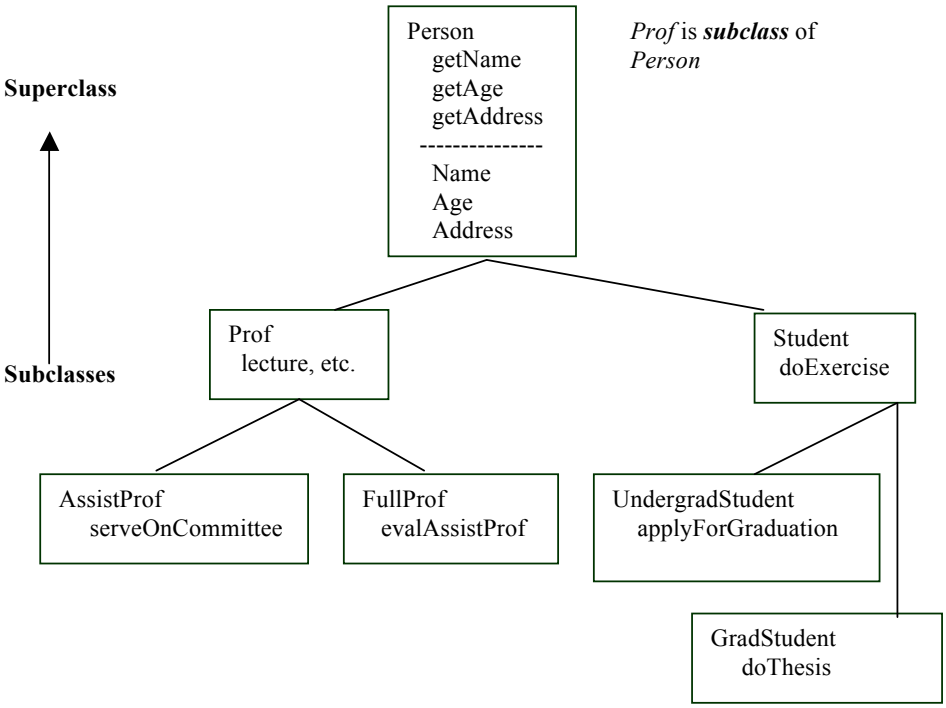
Levine/Eisman are ***instances*** of Profs

Jane, Armen, Helen, Levine, Eisman are all ***instances*** of their classes; they are ***OBJECTS capable of participating in a computation***

There are many objects in an OO Program: they communicate via ***message passing*** to accomplish tasks

Class Hierarchy

Organize classes in a structured fashion to deal with complexity



Class specifications (Roles, information/data) provided at higher levels (e.g. Person) are available for classes at lower levels (Prof): we reuse roles/data

e.g. if Armen is a GradStudent then we can pass him the message *getName* to get his name ➔ we look for the getName method in the GradStudent class, then Student, then Person (find it there so execute that method)

Note that each *Student* will be an instance of either *UndergradStudent* or *GradStudent*; each *Prof* will be an instance of *AssistProf* or *FullProf*

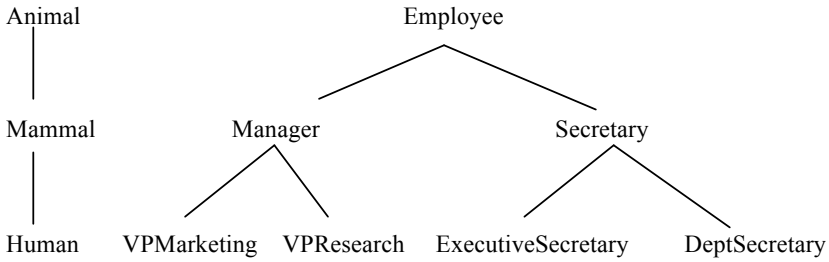
We use *Prof*, *Student*, and *Person* for specifying our hierarchy - e.g. we define roles and information that ALL *Person*s have in common in the *Person* class specification

➔ we will never directly create a *Person* instance

➔ these are ***ABSTRACT*** classes (used to specify common roles, behavior and data but will never create instances directly from these classes).

If we pass the message getName to FullProf and it's method is not specified therein, we continue searching for the method in its superclass (i.e. Prof, then Person) until we find it

Other hierarchies in our lives we use to deal with complexity:



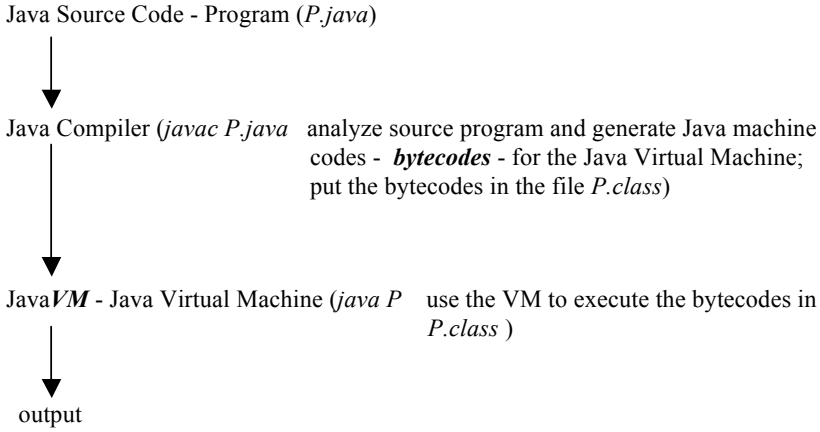
Reading:

Budd text: Chapter 1

Core Java 2: Chapter 1

III. Chapter 2 - A Java Perspective

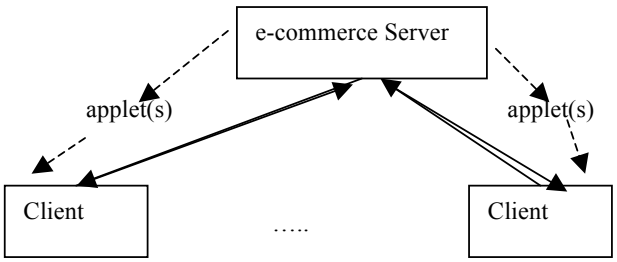
WRITE ONCE RUN ANYWHERE



Java***VM is a program that emulates the Java machine by INTERPRETING the bytecodes (i.e. scans the bytecodes and performs the indicated operations on the VM)***

Since the VM is a program it can execute the same bytecodes (P.class) on whatever machine it runs on - *write/compile the P.java program* ***once*** *and run its bytecodes* ***anywhere*** *the VM is running*

Client-Server computing:



Client uses, e.g., Netscape to browse to server, Server sends Java program (***Applet***) to be executed by the browser (Netscape has its own JavaVM for executing java bytecodes) The program might have a ***GUI*** (graphical user interface) for clients to use to type information (e.g. indicate products to purchase) which will be transmitted to the server (which can then send the purchased products via mail to the client and charge the client's credit card).

Do we want the server to send an Applet that might read client's private information and send to server? or delete client's files? or do other malicious activity?

NO!

The VM running in the client's browser interprets the bytecodes and doesn't allow codes that will access/modify client's files. It's like the applet is allowed to "*play*" in special *sandboxes* where it can't do harm. ***There are many security issues to deal with in this web environment***

WRITE ONCE RUN ANYWHERE

1. Compile the Java program (P.java) to bytecodes (P.class)
2. - Run on Window's machine
   - need VM program for Windows
   - it will run P.class OR
3. - Run on MAC machine
   - need VM program for MAC
   - it will run P.class OR
4. - Run on Linux machine
   - need VM program for Linux
   - it will run P.class

Therefore, the same P.class program is run (no recompilation is necessary) on different VM's

VM's provided by ,e.g., SUN, Symantec, Microsoft, …

SUN has VM's for their machines (UNIX), PC's, Macs

FROM text (page 22)

> *Java: A simple, object-oriented, network-savvy, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded, dynamic language*

Java is:

- Simple - no pointers, no multiple inheritance like C++
- Object-Oriented - everything is an object except primitive types like ints, floats,…
- Network savvy - knows about URL's sockets, etc.
- Interpreted - uses VM's
- Robust - built-in exception handling, garbage collection
- Secure - no pointers, applets use sandbox
- Architecture neutral - has its own VM
- Portable - its bytecodes don't depend on host machine, just on VM; machine dependant details (sizes of ints, floats) are fixed
- Multithreaded - parallel execution, client-server computing
- Dynamic - pluggable code and classes

Reading:

Budd text: Chapter 2

Core Java 2: Chapter 2

IV. Chapter 3 - Object Oriented Design

- Provide ***Reusable Software Components***: Software ***Integrated Circuits***
- Program for Change
- Focus on Data, not the tasks

  The data is less prone to change, unlike the tasks
- Think about the objects in a system and their responsibilities - ***what they will do;***

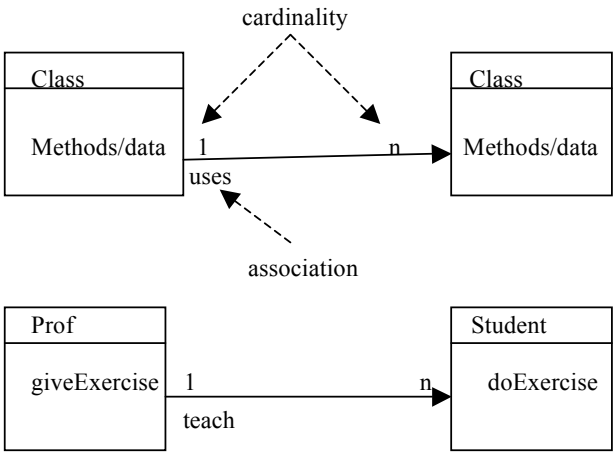
  NOT how they will do it
- People working in teams need to have well-established mechanisms for communication
  - Minimize surface area (the amount they need to know about each other's components in order to use them)
  - Minimize interactions between components
  - Each component has data and behavior (methods)
  - Encapsulate data (put data behind a wall)within a component to control access and ensure the integrity of the data

CRC Cards - Component, Responsibility, Collaborators

Component Name (Class)	**Collaborators**
Responsibility of component WHAT it does (methods)	List of other components it deals with (e.g. whose services it uses)

INDEX CARDS

OR



Class Diagram

1 Prof *teaches* n Students

Note: Class Diagrams provide the same information as CRC cards; there are ***CASE*** tools (Computer Aided Software Engineering) for constructing class diagrams - *Rational Rose*

Plan for Change

1. Localize changes so there aren't cascading changes to many components with slight changes of problem spec
2. Try to predict *tension points* - points in the code that are *likely to change* - code these areas carefully to minimize impacts of change
3. Minimize/isolate hardware dependencies (porting considerations)
4. Reduce coupling between components (this will help localize changes)
5. Document important design decisions to help future programmers understand (e.g. philosophy of design which might be manifested throughout system)

Software Components

Behavior = methods/protocol of class

State = information/data of class

e.g.

Student

study ← behavior

doExercises

---

coursesCompleted ← state

coursesEnrolledInThisTerm

Cohesion: the degree to which the responsibilities of a single component form a meaningful unit - *tasks* (behaviors) defined in the unit should be *strongly related*

Coupling: the relationship between software components - *reduce coupling* as much as possible to minimize cascading changes

Interfaces vs. Implementation

Other programmers needing to use my component only need to know what services my component offers ***NOT HOW*** the services are implemented - I can change the implementation of a service and the users' code is unaffected

e.g. Prof tells students to study (students manage the time, place, resources concerning their studies); Prof ***doesn't manage HOW*** the student is studying (too much for the Prof to handle; students can change HOW they study without affecting the Prof's job)

e.g. I might need to use a list to implement some part of my program; somebody else programs the list, giving capabilities to add items, remove items, etc. The other person can choose to use a linked list implementation or array or ?? My code should just use the services/behavior they provide without any knowledge as to how the list is implemented. If they subsequently decide to change their implementation then my code remains unchanged. **THIS IS AN EXTREMELY IMPORTANT CONCEPT**

Note that we ***ALWAYS PROGRAM TO AN INTERFACE*** without any knowledge of the implementation

> ***Interfaces specify what behavior is included without any details on how that behavior is implemented***

Parnas's Principles (from text)

- The developer of a software component must provide the intended user with all the information needed to make effective use of the services provided by the component, and should provide *no* other information.
- The developer of a software component must be provided with all the information necessary to carry out the given responsibilities assigned to the component, and should be provided with *no* other information.

Naming (text page 45)

- Use pronounceable names
- Use capitalization to mark the beginning of a new word within a name, such as *cardReader* rather than, e.g. *cardreader*
- Examine abbreviations carefully - use meaningful abbreviations (meaningful to more than just you)
- Avoid names with several interpretations. e.g. Does the *empty* function tell whether something is empty, or empty the values from the object?
- Avoid digits within a name. They are easy to misread as letters (0 as O)
- Name functions and variables that yield *Boolean* values so they describe clearly the interpretation of a *true* or *false* value; e.g. "PrinterIsReady" clearly indicates that a true value means the printer is working, whereas "PrinterStatus" is much less precise.
- Take extra care in the selection of names for operations that are costly and infrequently used. Doing so can avoid errors caused by using the wrong function.

Reading:

Budd text: Chapter 3

V. Chapter 4 – Some Notes on the Java Language

- Developed at *Sun Microsystems* 1995
- An ***Object Oriented*** Programming Language
- Based on the *C* programming language
- Evolved from *C++* and *Smalltalk*
- To provide a *ubiquitous* programming environment for ***web-based programming*** - **completely portable** system within a *distributed*programming environment on *heterogeneous* architectures
- **JDK's** currently provided free by *Sun Microsystems*

Compiling and Executing Java Programs

<Text Page 58>

```
import java.lang.*;          // this is saved in file SecondProgram.java
public class SecondProgram {
    public static void main ( String args[] ) {
            if (args.length > 0)   {
                    System.out.println("Hello " + args[0]);
             } else {
                    System.out.println( "Hello everybody!");
             }
      }
}
```

To compile: *javac SecondProgram.java*

To execute: *java SecondProgram* note absence of file extension

Output: Hello everybody!

or

To execute: java SecondProgram a b c

Output: Hello a

Notes:

1. *import* components from standard library
2. 1 *public* class per file; file name is same as this public class name - *SecondProgram.java*
3. *public static void main* - main program that takes an array of String args from user
4. *length* used to determine the number of elements in an array
5. + used to concatenate Strings
6. *System.out.println* used to print a String on a line followed by carriage-return/linefeed
7. use of curly braces, even when not needed to ensure against future errors
8. indentation, alignment of braces, etc.

A Note on the Default Constructor

If the programmer does not provide ANY constructors for a class then the compiler will include a *default, public, no-argument constructor*.

e.g.

```
public class A {
        int i;
        void p() {
                System.out.println(i);
        }
}
```

The compiler will provide:

```
        public A() {}
```

```
public class A {
        int i;
        private A() {...}   // no default constructor provided
        void p() {
                System.out.println(i);
        }
}
```

```
public class A {
        int i;
        public A(int j) {i = j;}   // no default constructor provided
        void p() {
                System.out.println(i);
        }
}
```

A Brief Note on Reference Semantics Used by Java

Unlike C++, in Java we can only refer to objects:

A a = new A(); // *a* refers to an instance of an *A*
A aa;
aa = a; // now, *a* and *aa* refer to the ***same object***;
B b = new B(5); // *b* has an instance variable *i* which is set to 5
aa.incI(b) // *b*'s variable *i* is bumped by 1
...

```
class A {
   ...
   public void incI(B bb) {
      // now, bb and b both refer to the same object; bb is not a copy of b!
      bb.inc();
   }
    ...
}
class B {
   private int i = 0;
   public int inc() {
      return ++i;
   }
   B(int i) {
      this.i = i;
   }
}
```

Exception Handling

Example raising an exception without catching it properly:

```
public class ExceptionTest1 {
 // generate array exception
    static int numbers[] = {0,-2,5,27,3};
    public static void main(String argv[]) {
        int sum = 0;
        for (int i = 0; i<= 5; i++) { // exception will occur in this loop
            sum += numbers[i];
  // attempt to access numbers[5] which is outside the array bounds
        }
    }}
```

Output:
C:\java ExceptionTest1
java.lang.ArrayIndexOutOfBoundsException

Example raising an exception and catching it properly:

```
public class ExceptionTest2 {
 // catch array exception properly
   static int numbers[] = {0,-2,5,27,3};
   public static void main(String argv[]) {
      int sum = 0;
      try {
         for (int i = 0; i<= 5; i++) {
            sum += numbers[i];
         }
      } catch (ArrayIndexOutOfBoundsException e) {
         System.out.println("***Array index out of bounds");
      }
      System.out.println("Sum: " + sum);
   }}
```

Output:
***Array index out of bounds
Sum: 33

Notes:
Exceptions are *out of the ordinary*, not necessarily errors
Syntax is:

```
try {
   <statementsA
} catch (Exception var1) {
   <statements to handle exception
}
...
} catch (Exception varn) {
   <statements to handle exception
} finally {
   <statements that will always be executed whether or not an exception occurred in
<statementsA
}
```

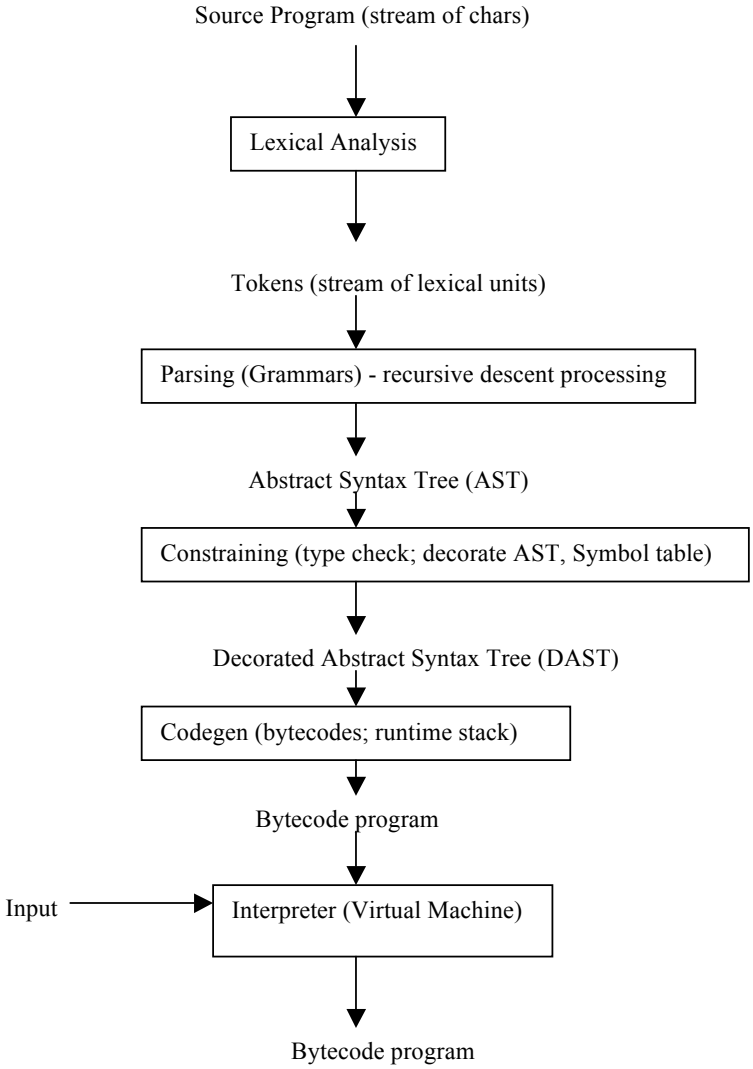
Sometimes an expression cannot be evaluated WITHOUT enclosing it in a *try clause.*

VI. A Comparision of Java and C++

Refer to http://unixlab.sfsu.edu/~levine/common/csc413/CppVsJava/

http://en.wikipedia.org/wiki/Comparison_of_Java_and_C%2B%2B
3/5/2011
Comparison_of_Java_and_C++.doc

VII. A Compiler



Extended Example

Source:

```
program { int  i int j
      i = i + j + 7                    ➔ lexical analyzer
      j = write(i)
}
```

Tokens:

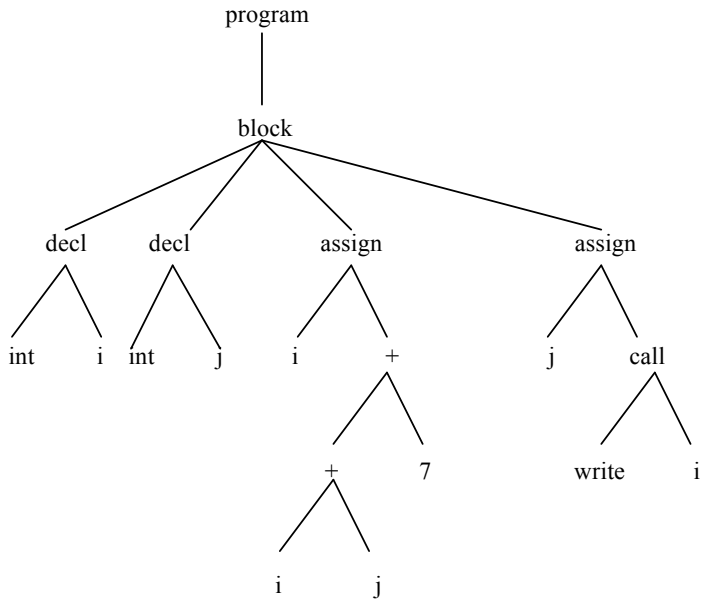
program leftBrace intType <id:i> intType <id:j>

<id:i> assign <id:i> plus <id:j> plus <int:7> ➔ parser

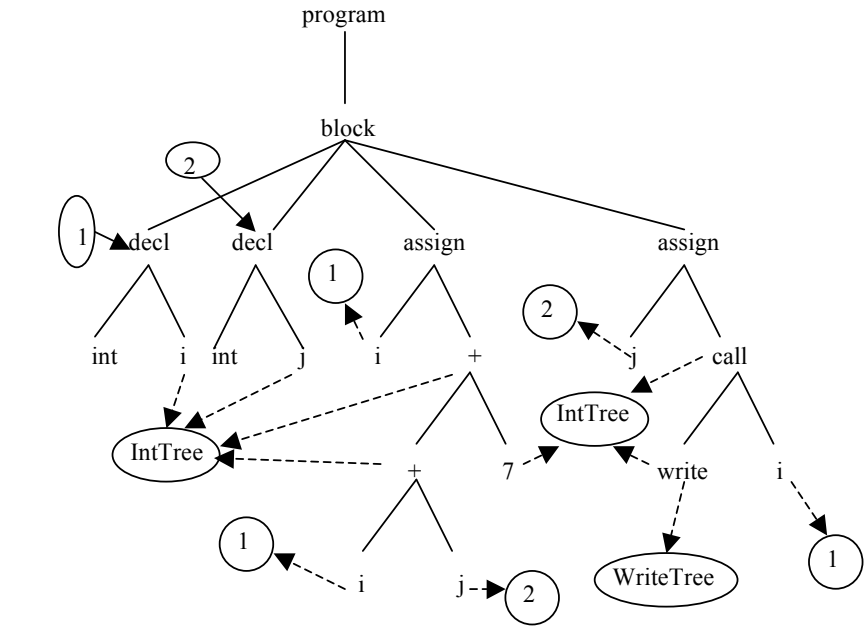
<id:j> assign <id:write> leftParen <id:i> rightParen

rightBrace

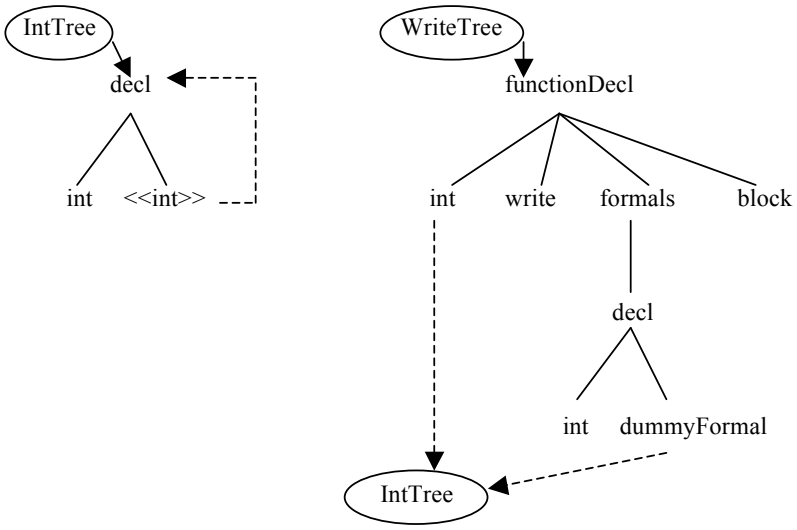
AST:



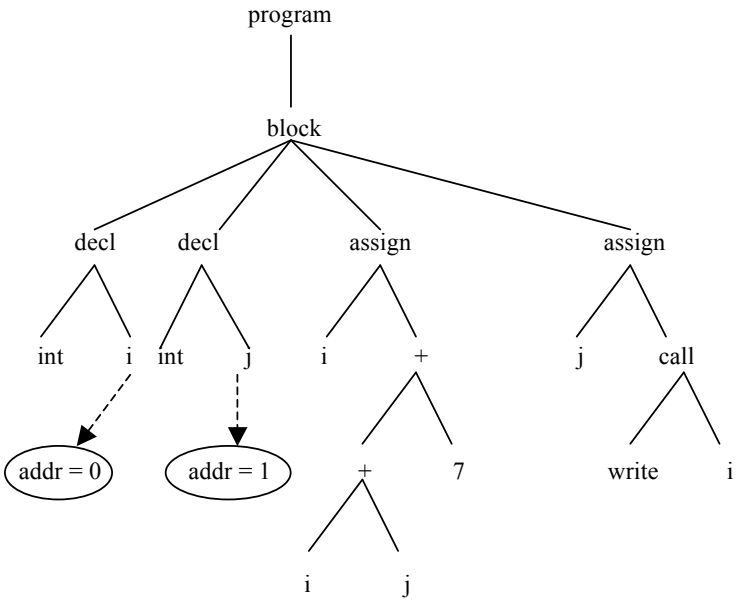
Decorated AST's:



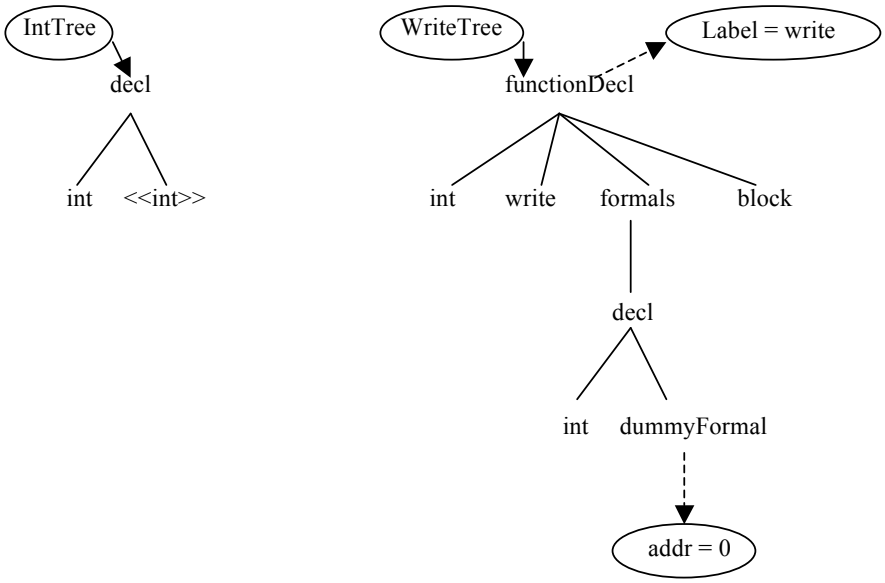
Intrinsic Trees:



Code generation:



Intrinsic Trees:



Executing the Program simple.x

```
program { int  i int j
      i = i + j + 7
      j = write(i)
}
```

Frame **(Activation Record)**	**Comments**
00	load init values of local variables (i and j) i is first, j is second
	generate code for i = i + j + 7
000	load i
0000	load j
000	add (sum is on top of Frame)
0007	load 7
007	add
70	store into i
	generate code for j = write(i)
707	load i
	ARGS 1 -- one arg for function
	Call *write*
70 | 7	Branch to write function; start new frame with arg(s) on top; in this case 7 is the only arg so it's the first slot in the new frame
70 | 77	load arg; write to output; value written remains on stack
707	return with value written as return value
77	store into j
--	clear local variables; *halt*

Bytecodes:

```
GOTO   start<<1>>
LABEL  read
READ                          Read function
RETURN
LABEL  write
LOAD   0   dummyFormal        Write function
WRITE
RETURN
LABEL  start<<1>>             i = i + j + 7
LIT 0     i                   init i to 0; i is offset 0 in current frame
LIT 0     j                   init j to 0; j is offset 1 in current frame
LOAD 0  i                     load variable at offset 0 (i)
LOAD 1  j
BOP +
LIT   7
BOP +
STORE  0  i                   store into i
LOAD   0  i                   j = write(i)
ARGS  1
CALL  write
STORE  1  j
POP  2                        remove local variables from stack
HALT
```

Complete Example of simple.x

```
program { int  i int j
      i = i + j + 7
      j = write(i)
}
```

TOKENS for simple.x

```
READLINE:   program { int i int j
      program            left: 0 right: 6
      {            left: 8 right: 8
      int            left: 10 right: 12
      i            left: 14 right: 14
      int            left: 16 right: 18
      j            left: 20 right: 20
READLINE:      i = i + j + 7
      i            left: 3 right: 3
      =            left: 5 right: 5
      i            left: 7 right: 7
      +            left: 9 right: 9
      j            left: 11 right: 11
      +            left: 13 right: 13
      7            left: 15 right: 15
READLINE:      j = write(i)
      j            left: 3 right: 3
      =            left: 5 right: 5
      write            left: 7 right: 11
      (            left: 12 right: 12
      i            left: 13 right: 13
      )            left: 14 right: 14
READLINE:   }
      }            left: 0 right: 0
```

AST for simple.x

```
1:  Program
2:    Block
5:      Decl
3:        IntType
4:        Id: i
8:      Decl
6:        IntType
7:        Id: j
10:     Assign
9:        Id: i
14:       AddOp: +
12:         AddOp: +
11:           Id: i
13:           Id: j
15:         Int: 7
17:     Assign
16:       Id: j
19:       Call
18:         Id: write
20:         Id: i
```

DECORATED AST for simple.x

```
1:  Program
2:    Block
5:      Decl
3:        IntType
4:        Id: i          Dec: 28                  28 is intTree
8:      Decl
6:        IntType
7:        Id: j          Dec: 28
10:     Assign
9:        Id: i          Dec: 5
14:       AddOp: +           Dec: 28
12:         AddOp: +           Dec: 28
11:           Id: i          Dec: 5
13:           Id: j          Dec: 8
15:         Int: 7           Dec: 28
17:     Assign
16:       Id: j          Dec: 8
19:       Call           Dec: 28
18:         Id: write          Dec: 35            35 is FunctionDecl tree for write
20:         Id: i          Dec: 5
```

AST AFTER CODEGEN for simple.x

```
1:  Program
2:    Block
5:      Decl  Label: i
3:        IntType
4:        Id: i          Dec: 28  Addr: 0          0 is offset in local frame
8:      Decl  Label: j
6:        IntType
7:        Id: j          Dec: 28  Addr: 1
10:     Assign
9:        Id: i          Dec: 5
14:       AddOp: +           Dec: 28
12:         AddOp: +           Dec: 28
11:           Id: i          Dec: 5
13:           Id: j          Dec: 8
15:         Int: 7           Dec: 28
17:     Assign
16:       Id: j          Dec: 8
19:       Call           Dec: 28
18:         Id: write          Dec: 35
20:         Id: i          Dec: 5

----------INTRINSIC TREES----------
----------READ/WRITE TREES----------
31: FunctionDecl  Label: Read              Node 31 is readTree
32:   IntType          Dec: 28
23:   Id: read
33:   Formals
34:   Block
35: FunctionDecl  Label: Write             Node 35 is writeTree
36:   IntType          Dec: 28
24:   Id: write
40:   Formals
37:     Decl
38:       IntType
39:       Id: dummyFormal          Dec: 28
41:   Block
----------INT/BOOL TREES----------
28: Decl                                   Node 28 is intTree
29:   IntType
30:   Id: <<int>>          Dec: 28
25: Decl                                   Node 25 is boolTree
26:   BoolType
27:   Id: <<bool>>           Dec: 25
```

Bytecodes for simple.x

```
GOTO start<<1>>
LABEL Read
READ
RETURN
LABEL Write
LOAD 0 dummyFormal
WRITE
RETURN
LABEL start<<1>>
LIT 0 i
LIT 0 j
LOAD 0 i
LOAD 1 j
BOP +
LIT 7
BOP +
STORE 0 i
LOAD 0 i
ARGS 1
CALL Write
STORE 1 j
POP 2
HALT
```

Program factorial.x

```
program {boolean j int i
  int factorial(int n) {
     if (n < 2) then
        { return 1 }
     else
        {return n*factorial(n-1) }
  }

  while (1==1) {
     i = write(factorial(read()))
  }

}
```

Main Compiler Data Structures

- Tokens
- Symbols
- Trees
- Vectors
- Arrays
- Tables
- Linked Lists
- Hashtables
- Stacks

VIII. Lexical Analysis

Read a stream of characters that make up the source program; create a stream of tokens (lexical units, lexemes) by combining the chars appropriately

e.g. the characters 't', 'h', 'e', 'n' will be combined to build the *then* token,

the characters '1', '2', '4', '7' will be combined to form an *integer* token with value of *1247*

Token Categories:

Reserved words: *program int boolean if then else while return*

Identifiers: <the same as Java identifiers>

Integers: <a sequence of digits>

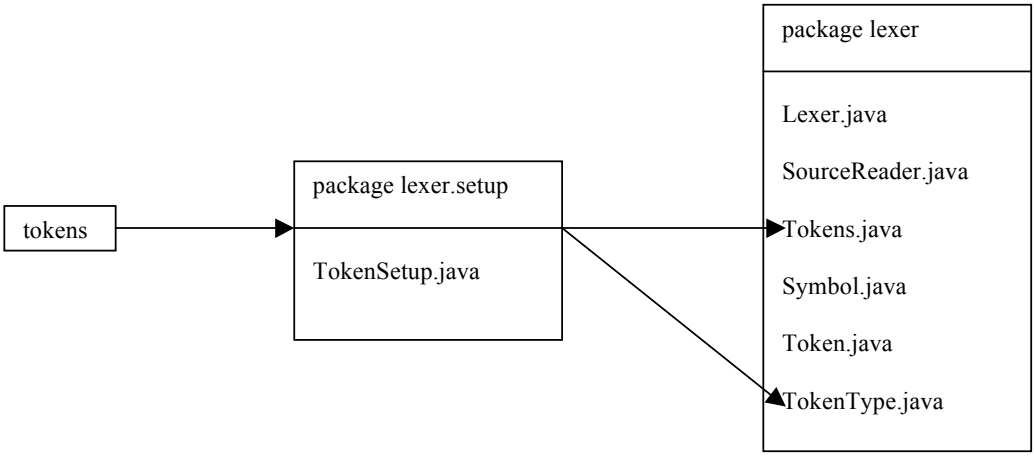
Operators: = == != < <= + - * / | &

Separators: { } ( ) ,

Comments: // until end-of-line

Whitespace: <spaces> <newlines> and other Java whitespace characters

Lexer Packages



tokens file

We will use a file *tokens for* listing the tokens:

Each line will have two strings

1. The Symbolic constant we will use in the compiler for the token

2. The actual token

tokens file:

Program	program
Int	int
BOOLean	boolean
If	if
Then	then
Else	else
While	while
Function	function
Return	return
Identifier	<id>
INTeger	<int>
LeftBrace	{
RightBrace	}
LeftParen	(
RightParen	)
Comma	,
Assign	=
Equal	==
NotEqual	!=
Less	<
LessEqual	<=
Plus	+
Minus	-
Or	|
And	&
Multiply	*
Divide	/
Comment	//

TokenSetup.java will read *tokens* and generate the files *Tokens.java* and *TokenType.java* - these files are based on *tokens*

The Tokens enum is actually a class; you can add methods, instance fields, as well as a constructor that can only be used to construct the enumeration values.

Values are accessed as Tokens.If , etc.

Tokens.java

```
package lexer;

/**
 *  This file is automatically generated<br>
 *  - it contains the enumberation of all of the tokens
*/
public enum Tokens {
  BogusToken,Program,Int,BOOLean,If,
    Then,Else,While,Function,Return,
    Identifier,INTeger,LeftBrace,RightBrace,LeftParen,
    RightParen,Comma,Assign,Equal,NotEqual,
    Less,LessEqual,Plus,Minus,Or,
    And,Multiply,Divide,Comment
}
```

TokenType.java

```
package lexer;

/**
 *  This file is automatically generated<br>
 *  it contains the table of mappings from token
 *  constants to their Symbols
*/
public class TokenType {
   public static java.util.HashMap<Tokens,Symbol> tokens = new
                java.util.HashMap<Tokens,Symbol>();
   public TokenType() {
     tokens.put(Tokens.Program,
              Symbol.symbol("program",Tokens.Program));
     tokens.put(Tokens.Int, Symbol.symbol("int",Tokens.Int));
     tokens.put(Tokens.BOOLean,
              Symbol.symbol("boolean",Tokens.BOOLean));
     tokens.put(Tokens.If, Symbol.symbol("if",Tokens.If));
     tokens.put(Tokens.Then, Symbol.symbol("then",Tokens.Then));
     tokens.put(Tokens.Else, Symbol.symbol("else",Tokens.Else));
     tokens.put(Tokens.While, Symbol.symbol("while",Tokens.While));
     tokens.put(Tokens.Function,
              Symbol.symbol("function",Tokens.Function));
     tokens.put(Tokens.Return, Symbol.symbol("return",Tokens.Return));
     tokens.put(Tokens.Identifier,
              Symbol.symbol("<id>",Tokens.Identifier));
     tokens.put(Tokens.INTeger,
              Symbol.symbol("<int>",Tokens.INTeger));
     tokens.put(Tokens.LeftBrace,
              Symbol.symbol("{",Tokens.LeftBrace));
     tokens.put(Tokens.RightBrace,
              Symbol.symbol("}",Tokens.RightBrace));
     tokens.put(Tokens.LeftParen,
              Symbol.symbol("(",Tokens.LeftParen));
     tokens.put(Tokens.RightParen,
              Symbol.symbol(")",Tokens.RightParen));
     tokens.put(Tokens.Comma, Symbol.symbol(",",Tokens.Comma));
     tokens.put(Tokens.Assign, Symbol.symbol("=",Tokens.Assign));
     tokens.put(Tokens.Equal, Symbol.symbol("==",Tokens.Equal));
     tokens.put(Tokens.NotEqual, Symbol.symbol("!=",Tokens.NotEqual));
     tokens.put(Tokens.Less, Symbol.symbol("<",Tokens.Less));
     tokens.put(Tokens.LessEqual,
              Symbol.symbol("<=",Tokens.LessEqual));
     tokens.put(Tokens.Plus, Symbol.symbol("+",Tokens.Plus));
     tokens.put(Tokens.Minus, Symbol.symbol("-",Tokens.Minus));
     tokens.put(Tokens.Or, Symbol.symbol("|",Tokens.Or));
     tokens.put(Tokens.And, Symbol.symbol("&",Tokens.And));
     tokens.put(Tokens.Multiply, Symbol.symbol("*",Tokens.Multiply));
     tokens.put(Tokens.Divide, Symbol.symbol("/",Tokens.Divide));
     tokens.put(Tokens.Comment, Symbol.symbol("//",Tokens.Comment));
   }
}
```

StringTokenizer class

When we read the *tokens* file we need to break each line into 2 Strings. We use the *StringTokenizer* class to break a string into tokens:

e.g.

```
StringTokenizer st = new StringTokenizer("these are several tokens");
while (st.hasMoreTokens()) {
   System.out.println(st.nextToken());
}
```

prints the following output:

```
these
are
several
tokens
```

Notes:

- *hasMoreTokens()* is used to ensure that before we try to get the next token (String) we actually have one available.
- *nextToken()* is used to obtain the next token (String); note we should always check that there are more items remaining prior to calling on *nextToken()* else we'll have problems

In general, when we have an *Enumeration* (a collection of Objects) we would like to sequence through it in a ***uniform and consistent*** fashion; we want a uniform and consistent ***interface*** for sequencing through various enumerations:

- *hasMoreElements()* checks whether we are done sequencing (we've seen all items)
- *nextElement()* will return the next element in the enumeration

We can enumerate through Vectors, arrays, sets, lists, as well as other collections

TokenSetup.java

```
package lexer.setup;

import java.util.*;
import java.io.*;

/**
 *  TokenSetup class is used to read the tokens from file tokens
 *  and automatically build the 2 classes/files TokenType.java
 *  and java <br>
 *  Therefore, if there is any change to the tokens then we only need to
 *  modify the file tokens  and run this program again before using the
 *  compiler
*/
public class TokenSetup {
   private String type, value; // token type/value for new token
   private int tokenCount = 0;
   private BufferedReader in;
   private PrintWriter table, symbols; // files used for new classes

   public static void main(String args[]) {
      new TokenSetup().initTokenClasses();
   }

   TokenSetup() {
      try {   // note that file separators depend on the O/S – Unix/DOS
       System.out.println("User's current working directory: " +
                                  System.getProperty("user.dir"));
       String sep = System.getProperty("file.separator");
         in = new BufferedReader( new FileReader("lexer" + sep + "setup" + sep +
                                       "tokens"));
         table = new PrintWriter(new FileOutputStream("lexer" + sep +
                                       "TokenType.java"));
         symbols = new PrintWriter(new FileOutputStream("lexer" + sep +
                                             "Tokens.java"));
      } catch (Exception e) { System.out.println(e); }
   }

/**
 *  read next line which contains token information;<br>
 *  each line will contain the token type used in lexical analysis and
 *  the printstring of the token: e.g.<br><ul>
 *  <li>Program program</li>
 *  <li>Int int</li>
 *  <li>BOOLean boolean</li></ul>
*/
   public void getNextToken() throws IOException {
      try {
         StringTokenizer st = new StringTokenizer(in.readLine());
         type = st.nextToken();
         value = st.nextToken();
      } catch (NoSuchElementException e) {
         System.out.println("***tokens file does not have 2 strings per line***");
```

```
                System.exit(1);
              } catch (NullPointerException ne) {
                // attempt to build new StringTokenizer when at end of file
                throw new IOException("***End of File***");
              }
              tokenCount++;
            }

          /**
           *  initTokenClasses will create the 2 files
           */
public void initTokenClasses() {
      table.println("package lexer;");
      table.println(" ");
      table.println("/**");
      table.println(" *  This file is automatically generated<br>");
      table.println(" *  it contains the table of mappings from token");
      table.println(" *  constants to their Symbols");
      table.println("*/");
      table.println("public class TokenType {");
      table.println("   public static java.util.HashMap< Tokens,Symbol> tokens " +
                  = new java.util.HashMap< Tokens,Symbol>();");
      table.println("   public TokenType() {");
      symbols.println("package lexer;");
      symbols.println(" ");
      symbols.println("/**");
      symbols.println(" *  This file is automatically generated<br>");
      symbols.println(" *  - it contains the enumberation of all of the tokens");
      symbols.println("*/");
      symbols.println("public enum Tokens {");
      symbols.print("    BogusToken");

      while (true) {
        try {
          getNextToken();
        } catch (IOException e) {break;}

        String symType = " Tokens." + type;

        table.println("     tokens.put(" + symType +
          ", Symbol.symbol(\"" + value + "\"," + symType + "));");

        if (tokenCount % 5 == 0) {
          symbols.print(",\n    "+ type);
        } else {
          symbols.print("," + type);
        }
      }

      table.println("   }");
      table.println("}");
      table.close();
      symbols.println("\n  }\n}");
      symbols.close();
      try {
        in.close();
```

```
        } catch (Exception e) {}
    }
}
```

SourceReader.java

```
package lexer;

import java.io.*;

/**
 *  This class is used to manage the source program input stream;
 *  each read request will return the next usable character; it
 *  maintains the source column position of the character
*/
public class SourceReader {
   private BufferedReader source;
   private int lineno = 0,   // line number of source program
      position;     // position of last character processed
   private boolean isPriorEndLine = true;
                  // if true then last character read was newline so read in the next line
   private String nextLine;
/*
   public static void main(String args[]) {
      SourceReader s = null;
      try {
         s = new SourceReader("t");
         while (true) {
            char ch = s.read();
            System.out.println("Char: " + ch + " Line: " + s.lineno +
                  "position: " + s.position);
         }
      } catch (Exception e) {}

      if (s != null) {
         s.close();
      }
   }
*/

/**
 *  Construct a new SourceReader
 *  @param sourceFile the String describing the user's source file
 *  @exception IOException is thrown if there is an I/O problem
*/
   public SourceReader(String sourceFile) throws IOException {
      source = new BufferedReader(new FileReader(sourceFile));
   }

   void close() {
      try {
         source.close();
      } catch (Exception e) {}
   }

/**
 *  read next char; track line #, character position in line<br>
 *  return space for newline
 *  @return the character just read in
```

```
 *  @IOException is thrown for IO problems such as end of file
*/
   public char read() throws IOException {
      if (isPriorEndLine) {
         lineno++;
         position = -1;
         nextLine = source.readLine();
         if (nextLine != null) {
            System.out.println("READLINE:   "+nextLine);
         }
         isPriorEndLine = false;
      }
      if (nextLine == null) {  // hit eof or some I/O problem
         throw new IOException();
      }
      if ( nextLine.length() == 0) {
         isPriorEndLine = true;
         return ' ';
      }
      position++;
      if (position >= nextLine.length()) {
         isPriorEndLine = true;
         return ' ';
      }
      return nextLine.charAt(position);
   }

/**
 *  @return the position of the character just read in
*/
   public int getPosition() {
      return position;
   }

/**
 *  @return the line number of the character just read in
*/
   public int getLineno() {
      return lineno;
   }
}
```

HashMap *(from http://java.sun.com/j2se/1.4.2/docs/api/index.html)*

public interface ***Map***

An object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value.

public abstract class ***AbstractMap*** *extends Object implements Map*

This class provides a skeletal implementation of the Map interface, to minimize the effort required to implement this interface.

public class ***HashMap*** *extends AbstractMap implements Map, Cloneable, Serializable*

Hash table based implementation of the Map interface. This implementation provides all of the optional map operations, and permits null values and the null key. (The HashMap class is roughly equivalent to Hashtable

Methods in ***HashMap*** include:

public abstract Object get(Object) -- retrieve an object with the given key
public Set **keySet**() - Returns a set view of the keys contained in this map.
public abstract Object put(Object key,Object value) -- enter the key/value pair

Returns:
The previous value to which the key was mapped in this HashMap, or null if the key did not have a previous mapping.

This class implements a hash table data structure, which maps keys to values. Any non-null object can be used as a key or as a value. To successfully store and retrieve objects from a ***HashMap***, the objects used as keys must implement the *hashCode* method and the *equals* method. In our case, the keys are Strings, which implement *hashCode* and *equals*

Symbol class

Hashtables are used with Symbols

Symbols:

- String and Type of Token
- All Strings (corresponding to tokens) found in the source program will be in the hashtable structure which is part of the Symbol class
- Before we begin we'll place all Tokens in the Symbol hashtable
- Each String will be inserted **exactly once**

e.g.

```
program { int j  int k
  j = j + k
}
```

Each Token has 4 pieces of information: 1. String of Token 2. Token type 3. starting column and 4. ending column; We'll group the first 2 items as a Symbol object (new class)

```
Token (Symbol ("program" , Tokens .Program),1,7)
Token (Symbol ("{" , Tokens .LeftBrace),9,9)
Token(Symbol("int" , Tokens .Int),11,13)
Token(Symbol("j", Tokens .Identifier),15,15)
Token(Symbol("int", Tokens .Int) ,17,19)
Token(Symbol("k", Tokens .Identifier),21,21)
Token(Symbol("j", Tokens .Identifier),2,2)
Token(Symbol("=", Tokens .Assign),4,4)
Token(Symbol("j", Tokens .Identifier),6,6)
Token(Symbol("+", Tokens .Plus),8,8)
Token(Symbol("k", Tokens .Identifier),10,10)
Token(Symbol("}", Tokens .RightBrace),1,1)
```

Note the repeated Symbol objects; this is wasteful. Better to simply create a single Symbol object and reuse it for each occurrence

Symbol(String s, Tokens kind) -- insert s into the Hashtable with value given by kind; if the entry is already in the Hashtable then just return the entry

Note:

We use *Symbol("j", Tokens Identifier)* 3 times. For efficiency considerations we only want to create 1 instance of Symbol. We use a Hashtable to check if the Symbol has already been created. If so, re-use that instance. If not, create a new instance. All of this logic is encapsulated in the Symbol class.

Init: Prior to processing the user's program we'll create Symbol instances for all reserved words, operators, etc. so we'll find them later - e.g. Symbol("program", Tokens Program); <see TokenType.java>

Once lexer starts processing the user's program the only new symbols that will be created (added to the hashmap) will be id's and numbers – all other symbols would have been created before and re-used now.

Schema:

1. Insert all tokens in HashMap (HashMap <String,Symbol> maps Strings to Symbols):

   keys are Strings; *values* are *Symbol(key, type of symbol)*

   tokens.put(Tokens.Program, Symbol.symbol("program",Tokens.Program));

   symbols.put(newTokenString,s)

   symbols is the instance of the HashMap

   ➔ *symbols*.put("program", Symbol("program", *Tokens.Program*));

   symbols.get("program") yields Symbol("program", *Tokens.Program*)

2. Scan program; insert symbols not already in symbols (ids, ints); lookup symbols

Symbol. *symbols*: keys - Strings Values - Symbols(String + kind of String)

String	Symbol
"program"	("program",Tokens.Program)
"("	("(",Tokens.LeftParen)
…	…
"*"	("*",Tokens.Multiply)
…	…

If we lookup an id in *symbols*:

A. Reserved word - e.g. *program*

We find it and return *Symbol("program",Tokens.Program)*

B. User id not already in *symbols*

We don't find it, so we put a new entry and return the new Symbol

C. User id already in *symbols* (we return the entry that is retrieved)

If we lookup other tokens

A. Numbers -- put new entry, if not already there

B. Not found -- don't do anything

e.g. = vs. == vs. != / vs. // these are either one or two character tokens

if we have == we'll find it

if we have =a (i.e. *x =abc + y*) we won't find it so we just key on the first

character, the *equal*, and save the *a* for the start of the next token (the *abc* id)

Symbol.java

```
package lexer;

/**
 *  The Symbol class is used to store all user strings along with
 *  an indication of the kind of strings they are; e.g. the id "abc" will
 *  store the "abc" in name and Tokens.Identifier in kind
*/
public class Symbol {
  private String name;
  private Tokens kind;   // token kind of symbol

  private Symbol(String n, Tokens kind) {
    name=n;
    this.kind = kind;
  }

  // symbols contains all strings in the source program
  private static java.util.HashMap <String,Symbol> symbols
                          = new java.util.HashMap <String,Symbol> ();

  public String toString() {
            return name;
  }

  public Tokens getKind() {
    return kind;
  }

  /**
   * Return the unique symbol associated with a string.
   * Repeated calls to <tt>symbol("abc")</tt> will return the same Symbol.
   */
public static Symbol symbol(String newTokenString, Tokens kind) {
            Symbol s = symbols.get(newTokenString);
            if (s == null) {
                if (kind == Tokens.BogusToken) {  // bogus string so don't enter into symbols
                    return null;
                }
                //   System.out.println("new symbol: "+u+" Kind: "+kind);
               s = new Symbol(newTokenString,kind);
               symbols.put(newTokenString,s);
            }
            return s;
  }
}
```

Token.java

```
package lexer;

/** <pre>
 *  The Token class records the information for a token:
 *  1. The Symbol that describes the characters in the token
 *  2. The starting column in the source file of the token and
 *  3. The ending column in the source file of the token
 *  </pre>
*/
public class Token {
  private int leftPosition,rightPosition;
  private Symbol symbol;

/**
 *  Create a new Token based on the given Symbol
 *  @param leftPosition is the source file column where the Token begins
 *  @param rightPosition is the source file column where the Token ends
*/
  public Token(int leftPosition, int rightPosition, Symbol sym) {
    this.leftPosition = leftPosition;
    this.rightPosition = rightPosition;
    this.symbol = sym;
  }

  public Symbol getSymbol() {
    return symbol;
  }

  public void print() {
    System.out.println("       " + symbol.toString() +
                       "            left: " + leftPosition +
                       " right: " + rightPosition);
    return;
  }

  public String toString() {
    return symbol.toString();
  }

  public int getLeftPosition() {
    return leftPosition;
  }

  public int getRightPosition() {
    return rightPosition;
  }
/**
 *  @return the integer that represents the kind of symbol we have which
 *  is actually the type of token associated with the symbol
*/
  public Tokens getKind() {
    return symbol.getKind();
  }
}
```

```
/**
 *  newIdTokens are either ids or reserved words; new id's will be inserted
 *  in the symbol table with an indication that they are id's
 *  @param id is the String just scanned - it's either an id or reserved word
 *  @param startPosition is the column in the source file where the token begins
 *  @param endPosition is the column in the source file where the token ends
 *  @return the Token; either an id or one for the reserved word
*/
   public Token newIdToken(String id,int startPosition,int endPosition) {
      return
         new Token(startPosition,endPosition,Symbol.symbol(id,Tokens.Identifier));
   }
```

A. For an id: e.g. *newIdToken("abc", 1, 3)* ➔

Token(1,3,Symbol.symbol("abc",Tokens.Identifier))

If "abc" is already in *symbols* then its *Symbol* will be fetched; else a new *Symbol* object will be created/entered in *symbols*

B. For a reserved word: e.g. *newIdToken("program",5,11)* ➔

Token(5,11,Symbol.symbol("program",Tokens.Identifier))

The *Symbol* for "program" will be retrieved: symbol("program",0)

Schema for the operation of Lexer:

Get a char

If the char starts an id/reserved word

Get the rest of the id; create a new id token

If it's a reserved word then return the reserved word Token

If the char starts an int

Get all of the digits; insert the string in *symbols*

else

It's either a 1 or 2 char operator, separator, or comment (e.g. "+", "(", or "//") so

pick up the 2 chars and if they don't make up a valid token then try checking

against the first of the 2 chars; if this doesn't work then it's a bad char

Example: Simulate Lexer using:

program {int i i=i+2

symbols: (after the line above is scanned)

“int”	Symbol(“int”,Tokens.Int)
“program”	Symbol(“program”, Tokens.Program)
…	
“<id>”	Symbol(“<id>”, Tokens.Identifier)
“<int>”	Symbol(“<int>”, Tokens.INTeger)
“{“	Symbol(“{”, Tokens.LeftBrace)
…	
“=”	Symbol(“=”, Tokens.Assign)
“i”	Symbol(“i”, Tokens.Identifier)
“+”	Symbol(“+”, Tokens.Plus)
“2”	Symbol(“int”, Tokens.INTeger)
…	

Lexer.java

```
package lexer;


/**
 *  The Lexer class is responsible for scanning the source file
 *  which is a stream of characters and returning a stream of
 *  tokens; each token object will contain the string (or access
 *  to the string) that describes the token along with an
 *  indication of its location in the source program to be used
 *  for error reporting; we are tracking line numbers; white spaces
 *  are space, tab, newlines
*/
public class Lexer {
   private boolean atEOF = false;
   private char ch;     // next character to process
   private SourceReader source;

   // positions in line of current token
   private int startPosition, endPosition;

   public Lexer(String sourceFile) throws Exception {
      new TokenType();  // init token table
      source = new SourceReader(sourceFile);
      ch = source.read();
   }

/*
   public static void main(String args[]) {
      Token tok;
      try {
         Lexer lex = new Lexer("simple.x");
         while (true) {
            tok = lex.nextToken();
            String p = "L: " + tok.getLeftPosition() +
               " R: " + tok.getRightPosition() + "  " +
               TokenType.tokens.get(tok.getKind()) + " ";
            if ((tok.getKind() == Tokens.Identifier) ||
                (tok.getKind() == Tokens.INTeger))
               p += tok.toString();
            System.out.println(p + ": "+lex.source.getLineno());
         }
      } catch (Exception e) {}
   }
*/

/**
 *  newIdTokens are either ids or reserved words; new id's will be inserted
 *  in the symbol table with an indication that they are id's
 *  @param id is the String just scanned - it's either an id or reserved word
 *  @param startPosition is the column in the source file where the token begins
 *  @param endPosition is the column in the source file where the token ends
 *  @return the Token; either an id or one for the reserved words
```

```
*/
   public Token newIdToken(String id,int startPosition,int endPosition) {
      return new Token(startPosition,endPosition,Symbol.symbol(id,Tokens.Identifier));
   }

/**
 *  number tokens are inserted in the symbol table; we don't convert the
 *  numeric strings to numbers until we load the bytecodes for interpreting;
 *  this ensures that any machine numeric dependencies are deferred
 *  until we actually run the program; i.e. the numeric constraints of the
 *  hardware used to compile the source program are not used
 *  @param number is the int String just scanned
 *  @param startPosition is the column in the source file where the int begins
 *  @param endPosition is the column in the source file where the int ends
 *  @return the int Token
*/
   public Token newNumberToken(String number,int startPosition,int endPosition) {
      return new Token(startPosition,endPosition,
         Symbol.symbol(number,Tokens.INTeger));
   }

/**
 *  build the token for operators (+ -) or separators (parens, braces)
 *  filter out comments which begin with two slashes
 *  @param s is the String representing the token
 *  @param startPosition is the column in the source file where the token begins
 *  @param endPosition is the column in the source file where the token ends
 *  @return the Token just found
*/
   public Token makeToken(String s,int startPosition,int endPosition) {
      if (s.equals("//")) {  // filter comment
         try {
           int oldLine = source.getLineno();
           do {
              ch = source.read();
           } while (oldLine == source.getLineno());
         } catch (Exception e) {
              atEOF = true;
         }
         return nextToken();
      }
      Symbol sym = Symbol.symbol(s,Tokens.BogusToken);
      // be sure it's a valid token
      if (sym == null) {
          System.out.println("******** illegal character: " + s);
          atEOF = true;
          return nextToken();
      }
      return new Token(startPosition,endPosition,sym);
      }

/**
 *  @return the next Token found in the source file
*/
   public Token nextToken() { // ch is always the next char to process
      if (atEOF) {
```

```
   if (source != null) {
      source.close();
      source = null;
   }
   return null;
}
try {
   while (Character.isWhitespace(ch)) { // scan past whitespace
      ch = source.read();
   }
} catch (Exception e) {
   atEOF = true;
   return nextToken();
}
startPosition = source.getPosition();
endPosition = startPosition - 1;

if (Character.isJavaIdentifierStart(ch)) {
   // return tokens for ids and reserved words
   String id = "";
   try {
      do {
         endPosition++;
         id += ch;
         ch = source.read();
      } while (Character.isJavaIdentifierPart(ch));
   } catch (Exception e) {
      atEOF = true;
   }
   return newIdToken(id,startPosition,endPosition);
}
if (Character.isDigit(ch)) {
   // return number tokens
   String number = "";
   try {
      do {
         endPosition++;
         number += ch;
         ch = source.read();
      } while (Character.isDigit(ch));
   } catch (Exception e) {
      atEOF = true;
   }
   return newNumberToken(number,startPosition,endPosition);
}

// At this point the only tokens to check for are one or two
// characters; we must also check for comments that begin with
// 2 slashes
String charOld = "" + ch;
String op = charOld;
Symbol sym;
try {
   endPosition++;
   ch = source.read();
   op += ch;
```

```
            // check if valid 2 char operator; if it's not in the symbol
            // table then don't insert it since we really have a one char
            // token
            sym = Symbol.symbol(op, Tokens.BogusToken);
            if (sym == null) {  // it must be a one char token
               return makeToken(charOld,startPosition,endPosition);
            }
            endPosition++;
            ch = source.read();
            return makeToken(op,startPosition,endPosition);
         } catch (Exception e) {}
         atEOF = true;
         if (startPosition == endPosition) {
            op = charOld;
         }
         return makeToken(op,startPosition,endPosition);
      }
   }
```

IX. Parsing - Syntax Analysis of the Token Stream Yielding the AST

Check that the tokens are assembled in the correct fashion according to the grammar (check the structure/syntax of user's program);

Build the ***Abstract Syntax Tree (AST)*** to represent the source program. The AST represents the source program in a form that is *convenient for subsequent processing*.

Grammar for X

```
PROGRAM -> ‘program’ BLOCK              ==> program

BLOCK -> ‘{‘ D* S* ‘}’                  ==> block

D -> TYPE NAME                          ==> decl
   -> TYPE NAME FUNHEAD BLOCK           ==> functionDecl

TYPE  -> ‘int’
         ->  ‘boolean’

FUNHEAD  -> '(' (D list ',')? ')'       ==> formals

S -> ‘if’ E ‘then’ BLOCK ‘else’ BLOCK   ==> if
  -> ‘while’ E BLOCK                    ==> while
  -> ‘return’ E                         ==> return
  -> BLOCK
  -> NAME ’=’ E                         ==> assign

E -> SE
  -> SE ‘==’ SE                         ==> =
  -> SE ‘!=’ SE                         ==> !=
  -> SE ‘<’  SE                         ==> <
  -> SE ‘<=’ SE                         ==> <=

SE  ->  T
      ->  SE ‘+’ T                      ==> +
      ->  SE’ –‘ T                      ==> -
      ->  SE ‘|’ T                      ==> or

T  -> F
   -> T ‘*’ F                           ==> *
   -> T ‘/’ F                           ==> /
   -> T ‘&’ F                           ==> and

F  -> ‘(‘ E ‘)’
   -> NAME
   -> <int>
   -> NAME '(' (E list ',')? ')'        ==> call

NAME  -> <id>
```

Notes:

- <id> stands for any valid identifier
- <int> stands for any integer
- rules without the tree description (e.g. *F->(E)*) indicate the corresponding tree is the tree built for the subpart on the right-hand-side - the *E* in this case
- items in single quotes are *tokens*

Parser Notes:

There will be ***one (recursive) procedure for each nonterminal*** to process the right-hand-side of each syntax rule; an AST subtree will be built to represent the right-hand-side (the tree to be built is described by the grammar); if the user's program doesn't conform to the grammar then a syntax error will be reported;

Note that the procedure for the *Program* rule will build the AST for the entire program.

Grammars

G = (N,T,P,Start)

N:

The set of ***Nonterminals*** used to describe the *main structural components* (syntactic categories) of the language (e.g. blocks, programs, expressions, …)

T:

The set of ***terminals*** which are the tokens used by the programmer and determined by Lexer

P:

Rewriting rules (***productions***) describing what N's can be rewritten to

e.g. in the rule *S -> if E then BLOCK else BLOCK*

S represents statements in the user program; *S* is the rule's ***left-hand-side (LHS);*** *if E then BLOCK else BLOCK* is the rule's ***right-hand-side (RHS);*** *S* is in the set N; the RHS is a *string of symbols* composed from those symbols in N and T; this category describes *if statements*

Start:

The ***special start symbol*** in N which is used to start the rewriting process (derivation); we can continue rewriting as long as the string has nonterminals; if our grammar is:

S -> aS | bBS | c B -> bB | b

A sample derivation is:

S ➔ bBS ➔ bbBS ➔ bbbS ➔bbbaS ➔ bbbaaS ➔ bbbaac

A Sample Derivation Using G:

PROGRAM ➔ program BLOCK ➔

program {D S } ➔

program { TYPE NAME S } ➔

program { int NAME S} ➔

program { int a S } ➔

program { int a NAME = E } ➔

program { int a a = E } ➔

program { int a a = SE } ➔

program { int a a = T } ➔

program { int a a = F } ➔

program { int a a = 5 }

Building AST's as Indicated by G

PROGRAM -> ‘program’ BLOCK ==> program
Build a program tree with one kid for the BLOCK tree

BLOCK -> ‘{‘ D* S* ‘}’ ==> block
Build a block tree with kids for the declarations and statements; there are zero or more D's and 0 or more S's

D -> TYPE NAME FUNHEAD BLOCK ==> functionDecl
Build a functionDecl tree with 4 kids for

1. Return type of the function TYPE
2. Name of the function NAME
3. Function formals FUNHEAD
4. Function body BLOCK

FUNHEAD -> '(' (D list ',')? ')' ==> formals
Build the tree describing the formals for the corresponding function declaration; there is one tree per formal declaration (D list ','); the D's are separated by commas (the *list* indicates this property)

F -> NAME '(' (E list ',')? ')' ==> call
Build a call tree with one kid for the function name and one kid for each actual argument expression *E*

ASTS Built from Source Programs

Source:

```
program { int  i int j
     i = i + j + 7                    ➔ lexical analyzer
     j = write(i)
}
```

Tokens:

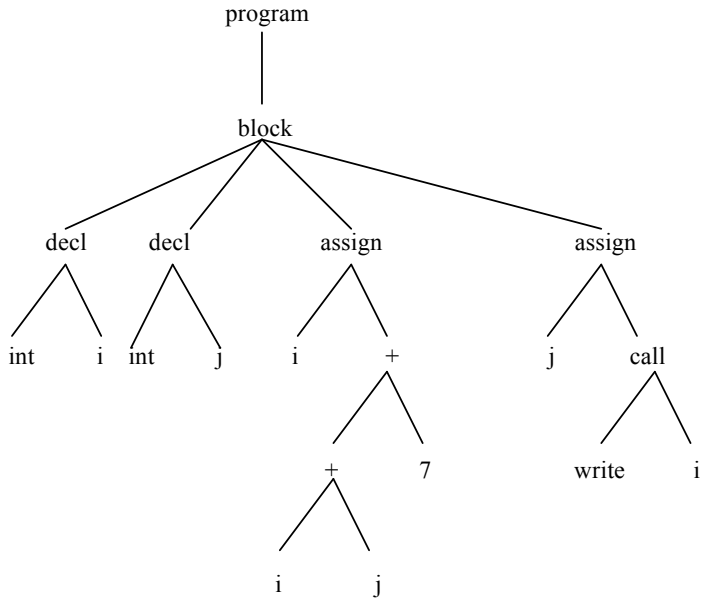
program leftBrace intType <id:i> intType <id:j>

<id:i> assign <id:i> plus <id:j> plus <int:7> ➔ parser

<id:j> assign <id:write> leftParen <id:i> rightParen

rightBrace

AST in 2-Dimensions:



AST in Printing Form

```
1:  Program
2:    Block
5:      Decl
3:        IntType
4:        Id: i
8:      Decl
6:        IntType
7:        Id: j
10:     Assign
9:        Id: i
14:       AddOp: +
12:         AddOp: +
11:           Id: i
13:           Id: j
15:         Int: 7
17:     Assign
16:       Id: j
19:       Call
18:         Id: write
20:         Id: i
```

Source Program:

```
program {boolean j int i
  int factorial(int n) {
     if (n < 2) then
        { return 1 }
     else
        {return n*factorial(n-1) }
  }
  while (1==1) {
     i = write(factorial(read()))
  }
}
```

----------AST----------

```
1:  Program
2:    Block
5:      Decl
3:        BoolType
4:        Id: j
8:      Decl
6:        IntType
7:        Id: i
11:     FunctionDecl
9:        IntType
10:       Id: factorial
12:       Formals
15:         Decl
13:           IntType
14:           Id: n
16:       Block
17:         If
19:           RelOp: <
18:             Id: n
20:             Int: 2
21:           Block
22:             Return
23:               Int: 1
24:           Block
25:             Return
27:               MultOp: *
26:                 Id: n
29:                 Call
28:                   Id: factorial
31:                   AddOp: -
30:                     Id: n
32:                     Int: 1
33:     While
35:       RelOp: ==
34:         Int: 1
36:         Int: 1
37:       Block
39:         Assign
38:           Id: i
41:           Call
40:             Id: write
43:             Call
```

```
42:         Id: factorial
45:         Call
44:           Id: read
```

The AST

Information at each node:

```
ArrayList<AST> kids;;       // kids of the current node
int nodeNum;                // used for identifying the node when printing
                            // (see printed AST's given above)
AST decoration              // used during the constraining phase
String label                // used during the code generation phase
```

Methods Included:

```
AST getKid(int i)           // return the ith kid of this node (kids are 1…)
int kidCount()              // number of kids from this node
ArrayList<AST>getKids()     // return a Collection of the kids of this node
AST addKid(AST kid)         // add a new kid to this node; return this node
```

Note: Each kind of AST will be described in a *subclass* of AST

java.util. ArrayList<E>

```
java.lang.Object
  └java.util.AbstractCollection<E>
      └java.util.AbstractList<E>
          └java.util.ArrayList<E>
```

```
public class ArrayList<E>
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
```

Resizable-array implementation of the `List` interface. (This class is roughly equivalent to `Vector...`

`public` **`ArrayList`**`()`
Constructs an empty list with an initial capacity of ten.

`public boolean` **`contains`**`(Object elem)`
Returns `true` if this list contains the specified element.

`public int` **`size`**`()`
Returns the number of elements in this list.

`public int` **`indexOf`**`(Object elem)`
Searches for the first occurence of the given argument, testing for equality using the `equals` method.

`public E` **`get`**`(int index)`
Returns the element at the specified position in this list.

`public boolean` **`add`**`(E o)`
Appends the specified element to the end of this list.

`public E` **`remove`**`(int index)`
Removes the element at the specified position in this list. Shifts any subsequent elements to the left (subtracts one from their indices).

Packages (Modules - basic system components)

Compiler packages:

Package	***Function of Package***
lexer	lexical analyzer - scan source program; output tokens
lexer.setup	automatically generate classes Sym and TokenType for lexer
parser	analyze tokens; check syntax; build AST
ast	Abstract Syntax Tree classes; representation of source program for efficient processing
visitor	used when walking (visiting) the AST - for printing, constraining, generating code
compiler	Compiler main program
constrain	visit the AST; check type constraints; decorate AST to hook up variable references to their declarations
codegen	visit the decorated AST; generate bytecodes
interpreter	Virtual machine, etc. used to execute the bytecodes generated for the source program

Compiling and Executing Java Packages

Consider the following directory structure on a PC (if UNIX is used then change the backslashes to forward slashes - change "\" to "/"):

d:
- d:\lexer
  - d:\lexer\setup
- d:\parser
- d:\ast
- d:\visitor
- d:\compiler
- d:\constrain
- d:\codegen
- d:\interpreter

To Compile:

cd d: Change directory to the root of the package structure -

javac lexer\setup.java* Compile all the *.java* (dot java) files in *d:\lexer\setup*

javac lexer.java* Compile all the *.java* files in *d:\lexer*

…

To Execute:

cd d:

java compiler.Compiler <arg> <arg> …

Execute the main program in *Compiler.java* with command line args

java compiler.Compiler factorial.x

AST.java

```
package ast;

import java.util.*;
import visitor.*;

/**
 *  The AST Abstract class is the Abstract Syntax Tree representation;
 *  each node contains<ol><li> references to its kids, <li>its unique node number
 *  used for printing/debugging, <li>its decoration used for constraining
 *  and code generation, and <li>a label for code generation</ol>
 *  The AST is built by the Parser
*/
public abstract class AST {
   protected ArrayList<AST> kids;
   protected int nodeNum;
   protected AST decoration;
   protected String label = "";   // label for generated code of tree

   static int NodeCount = 0;

   public AST() {
      kids = new ArrayList<AST>();
      NodeCount++;
      nodeNum = NodeCount;
   }

   public void setDecoration(AST t) {
      decoration = t;
   }

   public AST getDecoration() {
      return decoration;
   }

   public int getNodeNum() {
      return nodeNum;
   }

/**
 *  get the AST corresponding to the kid
 *  @param i is the number of the needed kid; it starts with kid number one
 *  @return the AST for the indicated kid
*/
   public AST getKid(int i) {
      if ( (i <= 0) || (i > kidCount())) {
         return null;
      }
      return kids.get(i - 1);
   }

/**
 *  @return the number of kids at this node
```

```
*/
   public int kidCount() {
      return kids.size();
   }

   public ArrayList<AST> getKids() {
      return kids;
   }

/**
 *  accept the visitor for this node - this method must be defined in each of
 *  the subclasses of AST
 *  @param v is the ASTVisitor visiting this node (currently, a printer,
 *  constrainer and code generator)
 *  @return the desired Object, as determined by the visitor
*/
   public abstract Object accept(ASTVisitor v);

   public AST addKid(AST kid) {
      kids.add(kid);
      return this;
   }

   public void setLabel(String label) {
      this.label = label;
   }

   public String getLabel() {
      return label;
   }

}


package ast;

import visitor.*;

public class ProgramTree extends AST {

   public ProgramTree() {
   }

   public Object accept(ASTVisitor v) {
      return v.visitProgramTree(this);
   }

}



package ast;

import lexer.Symbol;

public class IdTree extends AST {
```

```
   private Symbol symbol;
   private int frameOffset = -1;   // stack location for codegen

/**
 *  @param tok - record the symbol from the token Symbol
*/
   public IdTree(Token tok) {
      this.symbol = tok.getSymbol();
   }

   public Object accept(ASTVisitor v) {
      return v.visitIdTree(this);
   }

   public Symbol getSymbol() {
      return symbol;
   }

/**
 *  @param i is the offset for this variable as determined by the code generator
*/
   public void setFrameOffset(int i) {
      frameOffset = i;
   }

/**
 *  @return the frame offset for this variable - used by codegen
*/
   public int getFrameOffset() {
      return frameOffset;
   }

}



package ast;

import lexer.Symbol;

public class RelOpTree extends AST {
   private Symbol symbol;

/**
 *  @param tok contains the Symbol which indicates the specific relational operator
*/
   public RelOpTree(Token tok) {
      this.symbol = tok.getSymbol();
   }

   public Object accept(ASTVisitor v) {
      return v.visitRelOpTree(this);
   }

   public Symbol getSymbol() {
      return symbol;
```

```
    }
}
```

Simulate the parser with:

```
program {int i
        int f(int j) {int i return j+5}
        i = f(7)
}
```

Interface Iterator<E>

```
protected ArrayList<AST> kids;
…

Enumeration kidEnum = kids. elements();
```

`public interface` **`Iterator<E>`**

An iterator over a collection. Iterator takes the place of Enumeration in the Java collections framework. Iterators differ from enumerations in two ways:

- Iterators allow the caller to remove elements from the underlying collection during the iteration with well-defined semantics.
- Method names have been improved.

```
public abstract class ASTVisitor {

   public void visitKids(AST t) {
      for (AST kid : t.getKids()) {   // t.getKids() returns a collection of ArrayList<AST>
         kid.accept(this);
      }
      return;
   }
```

OR

```
public abstract class ASTVisitor {

   public void visitKids(AST t) {
      for (Iterator<AST> kidIterator =  t.getKids().iterator(); kidIterator.hasNext().;) {
         kidIterator.next().accept(this);
      }
      return;
   }
```

The Parser

```
/** <pre>
 *  Program -> 'program' block ==> program
 *  </pre>
 *  @return the program tree
 *  @exception SyntaxError - thrown for any syntax error
*/
   public AST rProgram() throws SyntaxError {
      AST t = new ProgramTree();
      expect( Tokens.Program);
      t.addKid(rBlock());
      return t;
   }
```

Notes:

1. Build the *ProgramTree*
2. Check that the next token is the *program* token and scan past it; report a *SyntaxError* if the scanned token doesn't match the *program* token
3. Call *rBlock* to check the syntax of the *block* that should follow and get the *BlockTree* that it returns; add the *BlockTree* as the kid of the *ProgramTree* just built

Consider the grammar describes the structure of all the phrases in the language (it's also known as a *phrase structure grammar*). Each nonterminal describes *the structure of a set of phrases*. For example, the 'S' nonterminal describes the set of *statement* phrases; the 'Program' nonterminal describes the high-level overall *program* structure.
Operation of each Parser method:

- When it is called, the token being scanned should be the start of one of the phrases described by the corresponding non-terminal (in this case, it's 'program')
- The method will check the stream of tokens derived from the user input stream to ensure they are syntactically correct, as described by the rule
- As the method checks the user token stream it advances the scanner. When it finishes scanning the phrase(s) for the method the scanner will be advanced to the first token JUST AFTER the phrase.
- As the method checks the stream of tokens for syntactic correctness it builds the AST as required; in this case, it builds an AST with root 'ProgramTree' and adds all the kids for the subphrases (this rule only has one type of phrase with only one subphrase – block).
- The method returns the AST just built
- If the method detects a syntax error it will throw a SyntaxError exception.

```
          /** <pre>
 *  block -> '{' d* s* '}'  ==> block
 *  </pre>
 *  @return  block tree
 *  @exception SyntaxError - thrown for any syntax error
 *        e.g. an expected left brace isn't found
*/
   public AST rBlock() throws SyntaxError {
      expect(Tokens.LeftBrace);
      AST t = new BlockTree();
      while (startingDecl()) {  // get decls
            t.addKid(rDecl());
      }
      while (startingStatement()) {  // get statements
            t.addKid(rStatement());
      }
      expect(Tokens.RightBrace);
      return t;
   }

   boolean startingDecl() {
      if (isNextTok(Tokens.Int) || isNextTok(Tokens.BOOLean)) {
         return true;
      }
      return false;
   }

   boolean startingStatement() {
      if (isNextTok(Tokens.If) || isNextTok(Tokens.While) || isNextTok(Tokens.Return)
            || isNextTok(Tokens.LeftBrace) || isNextTok(Tokens.Identifier)) {
         return true;
      }
      return false;
   }
```

Notes:

1. Check for the *left brace*; scan past it; else report *SyntaxError*
2. Next we expect **0 or more *d*'s** followed by **0 or more *s*'s**
3. Repeatedly check if the next token can start a ***D*** rule; if so, then we must find a ***D***
4. Do the same for the ***s*'s**
5. Check for the closing *right brace*

```
/** <pre>
 *  d -> type name                ==> decl
 *    -> type name funcHead block  ==> functionDecl
 *  </pre>
 *  @return either the decl tree or the functionDecl tree
 *  @exception SyntaxError - thrown for any syntax error
*/
   public AST rDecl() throws SyntaxError {
      AST t,t1;
      t = rType();
      t1 = rName();
      if (isNextTok( Tokens.LeftParen)) { // function; note naming of predicate
         // isNextTok does not scan past the LeftParen, it just checks for it
         t = (new FunctionDeclTree()).addKid(t).addKid(t1);
         // note cascading dot operator
         t.addKid(rFunHead());
         t.addKid(rBlock());
         return t;
      }
      t = (new DeclTree()).addKid(t).addKid(t1);
      return t;
   }
```

Notes:
Get the trees for *type* and *name*
Then, if we find a *left paren* we must be processing a *function declaration*

```
/** <pre>
 *  funHead  -> ‘ (‘  (decl list ',')? ‘ )’  ==> formals
 *  note a funhead is a list of zero or more decl's
 *  separated by commas, all in parens
 *  </pre>
 *  @return the formals tree describing this list of formals
 *  @exception SyntaxError - thrown for any syntax error
*/
   public AST rFunHead() throws SyntaxError {
      AST t = new FormalsTree();
      expect( Tokens.LeftParen);
      if (!isNextTok( Tokens.RightParen)) {
         do {
            t.addKid(rDecl());
            if (isNextTok( Tokens.Comma)) {
               scan();
            } else {
               break;
            }
         } while (true);
      }
      expect( Tokens.RightParen);
      return t;
   }
```

```
/** <pre>
 *     se  ->  t
 *         ->  se '+' t  ==> +
 *         ->  se '-' t  ==> -
 *         ->  se '|' t  ==> or
 *  This rule indicates we should pick up as many t 's as
 *  possible; the t 's will be left associative
 *  </pre>
 *  @return the tree corresponding to the adding expression
 *  @exception SyntaxError - thrown for any syntax error
*/
   public AST rSimpleExpr() throws SyntaxError {
      AST t, kid = rTerm();
      while ( (t = getAddOperTree()) != null) {
         t.addKid(kid);
         t.addKid(rTerm());
         kid = t;
      }
      return kid;
   }
```

Notes:
Consider the derivation with se:
se ➔ se + t ➔ se + t + t ➔ se + t + … + t + t ➔ t + t + … + t + t
Plus can be any of the adding operators (+ - |).
We want to find *any number* of ***t's*** with any of the *adding operators* between them. The adding operators should be performed in *left-to-right* order (*left associative*).

```
/** <pre>
 *     name  -> <id>
 *  </pre>
 *  @return the id tree
 *  @exception SyntaxError - thrown for any syntax error
*/
   public AST rName() throws SyntaxError {
      AST t;
      if (isNextTok( Tokens.Identifier)) {
         t = new IdTree(currentToken);
         scan();
         return t;
      }
      throw new SyntaxError(currentToken,Tokens.Identifier);
    }
```

Notes:
We have an IdTree; we must save the particular id (variable) with this tree

```
   private boolean isNextTok(Tokens kind) {
      if ((currentToken == null) || (currentToken.getKind() != kind)) {
         return false;
      }
      return true;
   }

   private void expect(Tokens kind) throws SyntaxError {
      if (isNextTok(kind)) {
```

```
            scan();
            return;
        }
        throw new SyntaxError(currentToken,kind);
    }

    private void scan() {
        currentToken = lex.nextToken();
        if (currentToken != null) {
            currentToken.print();   // debug printout
        }
        return;
    }
}
```

Notes:
We must be very careful to scan past a token as soon as we process it

```
class SyntaxError extends Exception {
    private Token tokenFound;
    private Tokens kindExpected;

/**
 *  record the syntax error just encountered
 *  @param tokenFound is the token just found by the parser
 *  @param kindExpected is the token we expected to find based on
 *  the current context
*/
    public SyntaxError(Token tokenFound, Tokens kindExpected) {
        this.tokenFound = tokenFound;
        this.kindExpected = kindExpected;
    }

    void print() {
        System.out.println("Expected: " +
          kindExpected);
        return;
    }
}
```

```
program { int  i int j
      i = i + j + 7
      j = write(i)
}
```

```
1:  Program
2:    Block
5:      Decl                    int i
3:        IntType
4:        Id: i
8:      Decl                    int j
6:        IntType
7:        Id: j
10:     Assign                  i = i + j + 7
9:        Id: i
14:       AddOp: +              Note left associativity of plus
12:         AddOp: +
11:           Id: i
13:           Id: j
15:         Int: 7
17:     Assign                  j = write(i)
16:       Id: j
19:       Call
18:         Id: write
20:         Id: i
```

Parser.java

```
package parser;

import java.util.*;
import lexer.*;
import ast.*;

/**
 *  The Parser class performs recursive-descent parsing; as a
 *  by-product it will build the <b>Abstract Syntax Tree</b> representation
 *  for the source program<br>
 *  Following is the Grammar we are using:<br>
 *  <pre>
 *  PROGRAM -> ‘program’ BLOCK ==> program
 *
 *  BLOCK -> ‘{‘ D* S* ‘}’  ==> block
 *
 *  D -> TYPE NAME                  ==> decl
 *    -> TYPE NAME FUNHEAD BLOCK      ==> functionDecl
 *
 *  TYPE  ->  ‘int’
 *        ->  ‘boolean’
 *
 *  FUNHEAD  -> '(' (D list ',')? ')'  ==> formals<br>
 *
 *  S -> ‘if’ E ‘then’ BLOCK ‘else’ BLOCK  ==> if
 *    -> ‘while’ E BLOCK              ==> while
 *    -> ‘return’ E                   ==> return
 *    -> BLOCK
 *    -> NAME ‘=’ E                   ==> assign<br>
 *
 *  E -> SE
 *    -> SE ‘==’ SE   ==> =
 *    -> SE ‘!=’ SE   ==> !=
 *    -> SE ‘<’  SE   ==> <
 *    -> SE ‘<=’ SE   ==> <=
 *
 *  SE  ->  T
 *      ->  SE ‘+’ T  ==> +
 *      ->  SE ‘-‘ T  ==> -
 *      ->  SE ‘|’ T  ==> or
 *
 *  T  -> F
 *     -> T ‘*’ F  ==> *
 *     -> T ‘/’ F  ==> /
 *     -> T ‘&’ F  ==> and
 *
 *  F  -> ‘(‘ E ‘)’
 *     -> NAME
 *     -> <int>
 *     -> NAME '(' (E list ',')? ')' ==> call<br>
 *
 *  NAME  -> <id>
 *  </pre>
```

```
*/
public class Parser {
   private Token currentToken;
   private Lexer lex;
   private EnumSet<Tokens> relationalOps =
           EnumSet.of(Tokens.Equal,Tokens.NotEqual,Tokens.Less,Tokens.LessEqual);
   private EnumSet<Tokens> addingOps =
           EnumSet.of(Tokens.Plus,Tokens.Minus,Tokens.Or);
   private EnumSet<Tokens> multiplyingOps =
           EnumSet.of(Tokens.Multiply,Tokens.Divide,Tokens.And);
/**
 *  Construct a new Parser;
 *  @param sourceProgram - source file name
 *  @exception Exception - thrown for any problems at startup (e.g. I/O)
*/
   public Parser(String sourceProgram) throws Exception {
      try {
         lex = new Lexer(sourceProgram);
         scan();
      }
       catch (Exception e) {
         System.out.println("********exception*******"+e.toString());
         throw e;
       };
   }

   public Lexer getLex() { return lex; }

/**
 *  Execute the parse command
 *  @return the AST for the source program
 *  @exception Exception - pass on any type of exception raised
*/
   public AST execute() throws Exception {
      try {
         return rProgram();
      }catch (SyntaxError e) {
         e.print();
         throw e;
         }
   }

/** <pre>
 *  Program -> 'program' block ==> program
 *  </pre>
 *  @return the program tree
 *  @exception SyntaxError - thrown for any syntax error
*/
   public AST rProgram() throws SyntaxError {
      // note that rProgram actually returns a ProgramTree; we use the
      // principle of substitutability to indicate it returns an AST
      AST t = new ProgramTree();
      expect(Tokens.Program);
      t.addKid(rBlock());
      return t;
   }
```

```
/** <pre>
 *  Block -> '{' d* s* '}'  ==> block
 *  </pre>
 *  @return  block tree
 *  @exception SyntaxError - thrown for any syntax error
 *         e.g. an expected left brace isn't found
*/
   public AST rBlock() throws SyntaxError {
      expect(Tokens.LeftBrace);
      AST t = new BlockTree();
      while (true) {  // get decls
        try {
         t.addKid(rDecl());
        } catch (SyntaxError e) { break; }
      }
      while (true) {  // get statements
        try {
         t.addKid(rStatement());
        } catch (SyntaxError e) { break; }
      }
      expect(Tokens.RightBrace);
      return t;
   }

/** <pre>
 *  d -> type name                ==> decl
 *    -> type name funcHead block  ==> functionDecl
 *  </pre>
 *  @return either the decl tree or the functionDecl tree
 *  @exception SyntaxError - thrown for any syntax error
*/
   public AST rDecl() throws SyntaxError {
      AST t,t1;
      t = rType();
      t1 = rName();
      if (isNextTok(Tokens.LeftParen)) { // function
         t = (new FunctionDeclTree()).addKid(t).addKid(t1);
         t.addKid(rFunHead());
         t.addKid(rBlock());
         return t;
      }
      t = (new DeclTree()).addKid(t).addKid(t1);
      return t;
   }

/** <pre>
 *  type  ->  'int'
 *  type  ->  'bool'
 *  </pre>
 *  @return either the intType or boolType tree
 *  @exception SyntaxError - thrown for any syntax error
*/
   public AST rType() throws SyntaxError {
      AST t;
      if (isNextTok(Tokens.Int)) {
```

```
            t = new IntTypeTree();
            scan();
        } else {
            expect(Tokens.BOOLean);
            t = new BoolTypeTree();
        }
        return t;
    }

/** <pre>
 *  funHead  ->  '( '  (decl list ',')?  ' )'  ==> formals
 *  note a funhead is a list of zero or more decl's
 *  separated by commas, all in parens
 *  </pre>
 *  @return the formals tree describing this list of formals
 *  @exception SyntaxError - thrown for any syntax error
*/
    public AST rFunHead() throws SyntaxError {
        AST t = new FormalsTree();
        expect(Tokens.LeftParen);
        if (!isNextTok(Tokens.RightParen)) {
            do {
                t.addKid(rDecl());
                if (isNextTok(Tokens.Comma)) {
                    scan();
                } else {
                    break;
                }
            } while (true);
        }
        expect(Tokens.RightParen);
        return t;
    }

/** <pre>
 *      S -> 'if' e 'then' block 'else' block  ==> if
 *        -> 'while' e block  ==> while
 *        -> 'return' e  ==> return
 *        -> block
 *        -> name '=' e  ==> assign
 *  </pre>
 *  @return the tree corresponding to the statement found
 *  @exception SyntaxError - thrown for any syntax error
*/
    public AST rStatement() throws SyntaxError {
        AST t;
        if (isNextTok(Tokens.If)) {
            scan();
            t = new IfTree();
            t.addKid(rExpr());
            expect(Tokens.Then);
            t.addKid(rBlock());
            expect(Tokens.Else);
            t.addKid(rBlock());
            return t;
        }
```

```
        if (isNextTok(Tokens.While)) {
            scan();
            t = new WhileTree();
            t.addKid(rExpr());
            t.addKid(rBlock());
            return t;
        }
        if (isNextTok(Tokens.Return)) {
            scan();
            t = new ReturnTree();
            t.addKid(rExpr());
            return t;
        }
        if (isNextTok(Tokens.LeftBrace)) {
            return rBlock();
        }
        t = rName();
        t = (new AssignTree()).addKid(t);
        expect(Tokens.Assign);
        t.addKid(rExpr());
        return t;
    }

/** <pre>
 *     e -> se
 *       -> se '==' se  ==> =
 *       -> se '!=' se  ==> !=
 *       -> se '<'  se  ==> <
 *       -> se '<=' se  ==> <=
 *  </pre>
 *  @return the tree corresponding to the expression
 *  @exception SyntaxError - thrown for any syntax error
*/
    public AST rExpr() throws SyntaxError {
        AST t, kid = rSimpleExpr();
        t = getRelationTree();
        if (t == null) {
            return kid;
        }
        t.addKid(kid);
        t.addKid(rSimpleExpr());
        return t;
    }

/** <pre>
 *     se  ->  t
 *         ->  se '+' t  ==> +
 *         ->  se '-' t  ==> -
 *         ->  se '|' t  ==> or
 *  This rule indicates we should pick up as many <i>t</i>'s as
 *  possible; the <i>t</i>'s will be left associative
 *  </pre>
 *  @return the tree corresponding to the adding expression
 *  @exception SyntaxError - thrown for any syntax error
*/
    public AST rSimpleExpr() throws SyntaxError {
```

```
      AST t, kid = rTerm();
      while ( (t = getAddOperTree()) != null) {
         t.addKid(kid);
         t.addKid(rTerm());
         kid = t;
      }
      return kid;
   }

/** <pre>
 *     t  -> f
 *        -> t '*' f  ==> *
 *        -> t '/' f  ==> /
 *        -> t '&' f  ==> and
 *  This rule indicates we should pick up as many <i>f</i>'s as
 *  possible; the <i>f</i>'s will be left associative
 *  </pre>
 *  @return the tree corresponding to the multiplying expression
 *  @exception SyntaxError - thrown for any syntax error
*/
   public AST rTerm() throws SyntaxError {
      AST t, kid = rFactor();
      while ( (t = getMultOperTree()) != null) {
         t.addKid(kid);
         t.addKid(rFactor());
         kid = t;
      }
      return kid;
   }

/** <pre>
 *     f  -> '(' e ')'
 *        -> name
 *        -> <int>
 *        -> name '(' (e list ',')? ')' ==> call
 *  </pre>
 *  @return the tree corresponding to the factor expression
 *  @exception SyntaxError - thrown for any syntax error
*/
   public AST rFactor() throws SyntaxError {
      AST t;
      if (isNextTok(Tokens.LeftParen)) { // -> (e)
         scan();
         t = rExpr();
         expect(Tokens.RightParen);
         return t;
      }
      if (isNextTok(Tokens.INTeger)) {  //  -> <int>
         t = new IntTree(currentToken);
         scan();
         return t;
      }
      t = rName();
      if (!isNextTok(Tokens.LeftParen)) {  //  -> name
         return t;
      }
```

```
        scan();     // -> name '(' (e list ',')? ) ==> call
        t = (new CallTree()).addKid(t);
        if (!isNextTok(Tokens.RightParen)) {
            do {
                t.addKid(rExpr());
                if (isNextTok(Tokens.Comma)) {
                    scan();
                } else {
                    break;
                }
            } while (true);
        }
        expect(Tokens.RightParen);
        return t;
    }

/** <pre>
 *     name  -> <id>
 *  </pre>
 *  @return the id tree
 *  @exception SyntaxError - thrown for any syntax error
*/
    public AST rName() throws SyntaxError {
        AST t;
        if (isNextTok(Tokens.Identifier)) {
            t = new IdTree(currentToken);
            scan();
            return t;
        }
        throw new SyntaxError(currentToken,Tokens.Identifier);
    }

    AST getRelationTree() {  // build tree with current token's relation
            Tokens kind = currentToken.getKind();
            if (relationalOps.contains(kind)) {
                    AST t = new RelOpTree(currentToken);
                    scan();
                    return t;
            } else {
                    return null;
            }
     }

    private AST getAddOperTree() {
            Tokens kind = currentToken.getKind();
            if (addingOps.contains(kind)) {
                    AST t = new AddOpTree(currentToken);
                    scan();
                    return t;
            } else {
                    return null;
            }
    }

    private AST getMultOperTree() {
            Tokens kind = currentToken.getKind();
```

```
            if (multiplyingOps.contains(kind)) {
                    AST t = new MultOpTree(currentToken);
                    scan();
                    return t;
            } else {
                    return null;
            }
    }

    private boolean isNextTok(Tokens kind) {
        if ((currentToken == null) || (currentToken.getKind() != kind)) {
            return false;
        }
        return true;
    }

    private void expect(Tokens kind) throws SyntaxError {
        if (isNextTok(kind)) {
            scan();
            return;
        }
        throw new SyntaxError(currentToken,kind);
    }

    private void scan() {
        currentToken = lex.nextToken();
        if (currentToken != null) {
            currentToken.print();   // debug printout
        }
        return;
    }
}

class SyntaxError extends Exception {
    /**
             *
             */
            private static final long serialVersionUID = 1L;
            /**
             *
             */
            private Token tokenFound;
    private Tokens kindExpected;

/**
 *  record the syntax error just encountered
 *  @param tokenFound is the token just found by the parser
 *  @param kindExpected is the token we expected to find based on
 *  the current context
*/
    public SyntaxError(Token tokenFound, Tokens kindExpected) {
        this.tokenFound = tokenFound;
        this.kindExpected = kindExpected;
    }

    void print() {
```

```
    System.out.println("Expected: " +
      kindExpected);
    return;
  }
```

Reading:

Core Java 2: Chapter 5

X. Tree Visitors

We need to walk the AST

1. To print it
2. To do type checking/decorating (constrainer)
3. To generate bytecodes (codegen)
4. Other

Assumption:

1. The structure of the tree is fixed (e.g. *Program* nodes will always have one kid)
2. We may have an unknown number of processors that may need to examine the tree (note the three items above)
3. There should be no code dealing with processing contained within the AST classes

   ➔ stabilize this code; don't keep changing the code whenever we decide we need a new processor (the AST should have no knowledge of the kind of processing that might eventually be needed)
4. The processors should not be responsible for walking the tree.
5. The processors need only know the types of nodes the tree contains - ***minimize coupling*** of processor objects to tree objects.

Schema

Each AST node is asked to ***accept*** a visitor (a processor walking the tree - e.g.

PrintVisitor is used to print the tree)

If the node accepts the visitor then the node calls on the visitor to visit the node's tree

and return any object of interest

e.g.

```
package ast;

import java.util.*;
import visitor.*;

public class ProgramTree extends AST {

   public ProgramTree() {
   }

   public Object accept(ASTVisitor v) {
      return v.visitProgramTree(this);
   }

}
```

The *ProgramTree* is asked to accept the visitor, *v*;

In our cases we don't need to check whether the visitor has constraints on visitation

rights, etc. so we simply allow *v* to visit the *ProgramTree*

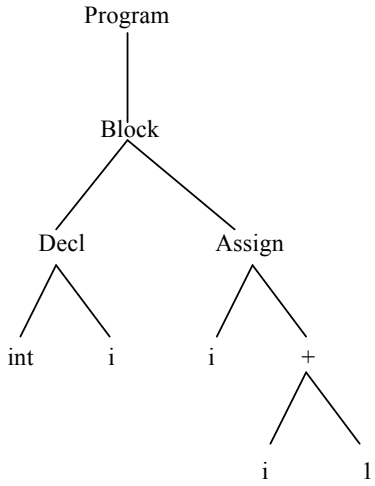
Each visitor must define procedures to visit every type of node (visitProgramTree,

visitBlockTree, etc.) - these methods perform relevant computations for the nodes they

are visiting.

PrintVisitor

t =




```
PrintVisitor pv = new PrintVisitor()
t.accept(pv)            // t is the ProgramTree above
  ProgramTree.accept(pv)
     pv.visitProgramTree(t)
          print("Program",t)
               print Program
                    visitKids(t)          // ask each kid to accept the visitor
                         Block.accept(pv)
```

Assignment and Dispatching

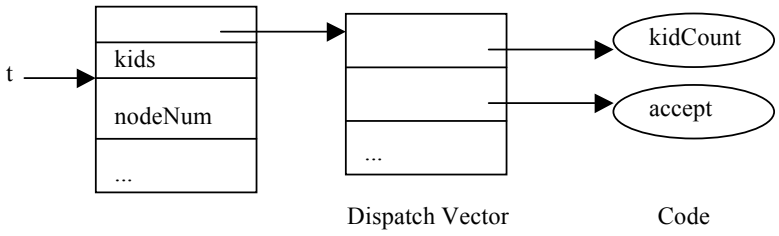
The compiler deduces an ***apparent type*** for each object by using the information in variable and method declarations (e.g. *AST t*; the apparent type of objects that *t* refers to is *AST*). Each object has an ***actual type*** that it receives *when it is created*: this is the type defined by the class that constructs it (e.g. *new RelOpTree());* the actual type herein is a *RelOpTree*). The compiler ensures the *apparent type* it deduces for an object is *always a supertype of the actual type* of the object. The compiler determines *what calls are legal based on the object's apparent type* (e.g. there is a *getSymbol* method in the *RelOpTree* class but not in the *AST* class; if we have *AST t = new RelOpTree();* we cannot use *t.getSymbol()* since the **apparent** class of *t* **does not contain** the *getSymbol* method). Dispatching causes method calls to go to the object's actual code - that is, the code provided by its class (e.g. in our case *.AST t = new RelOpTree();* if we subsequently include *t.accept(pv);* the *accept* method in the **actual** class of *t* - *RelOpTree* - will be called). This form of dispatching that can only be determined *dynamically* (we may not be able to determine which subclass *t* refers to until runtime and therefore which *accept* method to call until runtime) is also referred to as ***dynamic binding*** - we don't bind/associate a particular piece of code with the *accept* reference until runtime.

This type of dispatching can be implemented by each object containing (in addition to its instance variables) a pointer to a *dispatch vector*, which *contains pointers to the implementations of the object's methods*. Consider the following definitions in the compiler:

```
public abstract class AST {
   protected ArrayList<AST> kids;
   protected int nodeNum;
  ...
   public AST() {...}

   public int kidCount() { return kids.size(); }
  ...
   public abstract Object accept(ASTVisitor v);
```

```
}


public class ProgramTree extends AST {
   public ProgramTree() {}

   public Object accept(ASTVisitor v) { return v.visitProgramTree(this);  }

}

AST t = new ProgramTree();
t.accept();
/*
      the apparent type of t is an AST; this means we can access only those methods in
      the AST class such as kidCount() and accept();
      the actual type of the object t refers to is a ProgramTree; this means that t.accept()
      will be dispatched to the accept method in the ProgramTree class - the accept in
      t.accept() will be bound to the accept code in the ProgramTree class
*/
```

The following figure demonstrates the use of the dispatch table included with *t*:



Let's consider the following method once again, with respect to ***dynamic binding***:

```
   public Object accept(ASTVisitor v) { return v.visitProgramTree(this);  }
```

The issue of concern is ***which piece of code is associated with the method reference*** *visitProgramTree*. **IT IS NOT DETERMINABLE UNTIL RUNTIME WHICH CODE WILL BE ASSOCIATED WITH (BOUND TO) *visitProgramTree***. This results from the fact that *v* 's actual type is not determinable until runtime so we don't know whose *visitProgramTree* method will execute until runtime.

Double Dispatching

The visitor pattern employs a ***double dispatch*** mechanism which avoids the use of *switch* statements: We want to call the appropriate *visitor* method ==> we call the appropriate *accept* method which then calls the appropriate *visitor* method ==> ***NO SWITCH***

switch statements are dangerous since it is code that is *strongly coupled with the cases*. It is *prone to errors* when we consider more cases (we could easily forget to add the new cases to **all** the switch statements)

Dynamic Binding

- gives power to polymorphism
- is needed when the compiler determines that there is more than one possible method that could be executed by a particular call
- prevents programmers from having to write conditional statements to explicitly choose which code to run

Dynamic dispatch is the process of mapping a message to a specific sequence of code (method) at runtime - uses dynamic binding.

If the class of an object *t* is not known at compile-time, then *when t.accept()* is called, the program must decide at runtime which implementation of accept() to invoke, based on the runtime type of object *t*. This case is known as single dispatch because an implementation is chosen based on a single type—that of the *this* object.

Double dispatch *is the ability to dynamically select a method not only according to the run-time type of the receiver (single dispatch), but also according to the run-time type of the argument.* Thus, dynamic binding is applied *twice* - it's very powerful! It enhances re-usability and separation of responsibilities.

Consider our use of double dispatch using the Visitor design pattern:

```
PrintVisitor pv = new PrintVisitor()
t.accept(pv)        // t is some AST – let's assume we do not know which subtype of AST
                    // it is determined at RUNTIME which accept method code to execute
                    // dynamic dispatch is used – the code to execute is determined based
                    // on the actual class that t references; suppose t references a
                    // ProgramTree object
  ProgramTree.accept(pv)
                     // now, we consider the accept method in ProgramTree; it takes an
                     // ASTVisitor as argument; again, we use dynamic dispatch to
                     // determine at RUNTIME which visitProgramTree method to
                     // execute
      pv.visitProgramTree(t)
            print("Program",t)
                  print Program
                        visitKids(t)          // ask each kid to accept the visitor
                             Block.accept(pv)
```

ASTVisitor.java

```
package visitor;

import ast.*;
import java.util.*;

/**
 *  ASTVisitor class is the root of the Visitor hierarchy for visiting
 *  various AST's; each visitor asks each node in the AST it is given
 *  to accept  its visit; <br>
 *  each subclass <b>must</b> provide all of the visitors mentioned
 *  in this class; <br>
 *  after visiting a tree the visitor can return any Object of interest<br>
 *  e.g. when the constrainer visits an expression tree it will return
 *  a reference to the type tree representing the type of the expression
*/
public abstract class ASTVisitor {

   public void visitKids(AST t) {
      for (AST kid : t.getKids()) {
         kid.accept(this);
      }
      return;
   }

   public abstract Object visitProgramTree(AST t);
   public abstract Object visitBlockTree(AST t);
   public abstract Object visitFunctionDeclTree(AST t);
   public abstract Object visitCallTree(AST t);
   public abstract Object visitDeclTree(AST t);
   public abstract Object visitIntTypeTree(AST t);
   public abstract Object visitBoolTypeTree(AST t);
   public abstract Object visitFormalsTree(AST t);
   public abstract Object visitActualArgsTree(AST t);
   public abstract Object visitIfTree(AST t);
   public abstract Object visitWhileTree(AST t);
   public abstract Object visitReturnTree(AST t);
   public abstract Object visitAssignTree(AST t);
   public abstract Object visitIntTree(AST t);
   public abstract Object visitIdTree(AST t);
   public abstract Object visitRelOpTree(AST t);
   public abstract Object visitAddOpTree(AST t);
   public abstract Object visitMultOpTree(AST t);
}
```

PrintVisitor.java

```
package visitor;

import ast.*;

/**
 *  PrintVisitor is used to visit an AST and print it using
 *  appropriate indentation:<br>
 *  <pre>
 *  1. root
 *  2.   Kid1
 *  3.   Kid2
 *  4.     Kid21
 *  5.     Kid22
 *  6.     Kid23
 *  7.   Kid3
 *  </pre>
*/
public class PrintVisitor extends ASTVisitor {
   private int indent = 0;

   private void printSpaces(int num) {
      String s = "";
      for (int i = 0; i < num; i++) {
         s += ' ';
      }
      System.out.print(s);
   }

/**
 *  Print the tree
 *  @param s is the String for the root of t
 *  @param t is the tree to print - print the information
 *  in the node at the root (e.g. decoration) and its kids
 *  indented appropriately
*/
   public void print(String s,AST t) {
      // assume less than 1000 nodes; no problem for csc 413
      int num = t.getNodeNum();
      AST decoration = t.getDecoration();
      int decNum = (decoration == null)? -1 : decoration.getNodeNum();
      String spaces = "";
      if (num < 100) spaces += " ";
      if (num < 10) spaces += " ";
      System.out.print(num + ":" + spaces);
      printSpaces(indent);
      if (decNum != -1) {
         s += "          Dec: " + decNum;
      }
      String lab = t.getLabel();
      if ( lab.length() > 0 ) {
         s += "  Label: "+t.getLabel();
      }
```

```
        if (t.getClass() == IdTree.class) {
            int offset = ((IdTree)t).getFrameOffset();
            if (offset >= 0) {
                s += "  Addr: " + offset;
            }
        }
        System.out.println(s);
        indent += 2;
        visitKids(t);
        indent -= 2;
    }

    public Object visitProgramTree(AST t) { print("Program",t);  return null; }
    public Object visitBlockTree(AST t) { print("Block",t);  return null; }
    public Object visitFunctionDeclTree(AST t) { print("FunctionDecl",t);  return null; }
    public Object visitCallTree(AST t) { print("Call",t);  return null; }
    public Object visitDeclTree(AST t) { print("Decl",t);  return null; }
    public Object visitIntTypeTree(AST t) { print("IntType",t);  return null; }
    public Object visitBoolTypeTree(AST t) { print("BoolType",t);  return null; }
    public Object visitFormalsTree(AST t) { print("Formals",t);  return null; }
    public Object visitActualArgsTree(AST t)
                { print("ActualArgs",t);  return null; }
    public Object visitIfTree(AST t) { print("If",t);  return null; }
    public Object visitWhileTree(AST t) { print("While",t);  return null; }
    public Object visitReturnTree(AST t) { print("Return",t);  return null; }
    public Object visitAssignTree(AST t) { print("Assign",t);  return null; }
    public Object visitIntTree(AST t) {
        print("Int: " + ((IntTree)t).getSymbol().toString(),t);
        return null; }
    public Object visitIdTree(AST t) {
        print("Id: "+((IdTree)t).getSymbol().toString(),t);
        return null; }
    public Object visitRelOpTree(AST t) {
        print("RelOp: "+((RelOpTree)t).getSymbol().toString(),t);
        return null; }
    public Object visitAddOpTree(AST t) {
        print("AddOp: "+((AddOpTree)t).getSymbol().toString(),t);
        return null; }
    public Object visitMultOpTree(AST t) {
        print("MultOp: "+((MultOpTree)t).getSymbol().toString(),t);
        return null; }
}
```

Compiler.java

```
package compiler;

import ast.*;
import parser.Parser;
import java.util.*;
import constrain.Constrainer;
import codegen.*;
import visitor.*;

/**
 *  The Compiler class contains the main program for compiling
 *  a source program to bytecodes
*/
public class Compiler {

/**
 * The Compiler class reads and compiles a source program
*/
   public Compiler(String sourceFile) {
      try {
         System.out.println("----------TOKENS----------");
         Parser parser = new Parser(sourceFile);
         AST t = parser.execute();
         System.out.println("----------AST----------");
         PrintVisitor pv = new PrintVisitor();
         t.accept(pv);
/*  COMMENT CODE FROM HERE UNTIL THE CATCH CLAUSE WHEN
TESTING PARSER  */
         Constrainer con = new Constrainer(t,parser);
         con.execute();
         System.out.println("----------DECORATED AST----------");
         t.accept(pv);
/*  COMMENT CODE FROM HERE UNTIL THE CATCH CLAUSE WHEN
TESTING CONSTRAINER */
         Codegen generator = new Codegen(t);
         Program program = generator.execute();
         System.out.println("----------AST AFTER CODEGEN----------");
         t.accept(pv);
         System.out.println("----------INTRINSIC TREES----------");
         System.out.println("----------READ/WRITE TREES----------");
         con.readTree.accept(pv);
         con.writeTree.accept(pv);
         System.out.println("----------INT/BOOL TREES----------");
         con.intTree.accept(pv);
         con.boolTree.accept(pv);
         program.printCodes(sourceFile + ".cod");
         // if the source file is "abc" print bytecodes to abc.cod
      }catch (Exception e) {
         System.out.println("********exception*******"+e.toString());
       };
   }
```

```
  public static void main(String args[]) {
    if (args.length == 0) {
      System.out.println(
            "***Incorrect usage, try: java compiler.Compiler <file>");
      System.exit(1);
    }
    new Compiler(args[0]);
  }
}
```

Reading:

Core Java 2: Chapter 6, 11

XI. Chapter 8 - Inheritance

Def. (page 123)

The property that instances of a child class (or subclass) can access both data and behavior (methods) associated with a parent class (or superclass).

The ***Object*** class:

Is at the root of the inheritance **HIERARCHY**

==> EVERY CLASS HAS OBJECT AS ITS (IMPLICIT) PARENT

e.g. the following class definitions are equivalent:

```
class XYZ {                    class XYZ  extends Object{
...                            ...
}                              }
```

Defines the ***structure and behavior found in all classes***

Object

equals -- often overridden in subclasses

getClass() -- return the class of the receiver

```
if (t.getClass() == IdTree.class) {
```

hashCode() -- works in tandem with *equals*; if the programmer overrides *equals* then *hashCode* should be overridden (i.e. if *a.equals(b)*, then *a* and *b* must have the same hash code

toString() -- returns the String representation of the current Object

Subclass, SubTypes and Substitutability

AST t = parser.execute()

parser.execute() can return ***any type of AST***

Substitutability

From text:

1. Instances of the subclass must possess all data fields associated with the parent class
2. Instances of the subclass must implement, through inheritance at least (if not explicitly overridden) all functionality defined for the parent class. (They can also define new functionality but that is unimportant for this argument)
3. Thus, an instance of a child class can mimic the behavior of the parent class and should be ***indistinguishable*** from an instance of the parent class if substituted in a similar situation.

Subtype

It is used to describe the relationship between types that explicitly recognizes the *principle of substitution*. That is, a type B is considered to be a **subtype** of A if two conditions hold. The first is that an instance of B can legally be assigned to a variable declared as type A. And the second is that this value can then be used by the variable with ***no observable change in behavior***. A subtype preserves the meaning (purpose, or intent) of the parent.

Problem, *meaning* is extremely difficult to define. Think about how to define the LIFO characteristics of the stack.

Subclass

The mechanics of constructing a new class using inheritance, and is easy to recognize from the source description of a program by the presence of the keyword ***extends***. The

subtype relationship is more abstract, and is only loosely documented directly by the program source.

Type

Set of values, interface and properties

e.g.

A: *Integers*: Values (minint..maxint), Interface (operators such as + - …), Properties (property of division ➔ 8/5 = 1; we use truncation to assure the result is in the set of Integers; property of addition where 3+5 = 8)

B: Consider the *Stack* interface and a *strange* implementation

```
interface Stack {
        public void push(Object value);
        public Object top();
        public void pop();
}


class NonStack implements Stack {
        public void push(Object value) { v=value; }
        public Object top() { return v; }
        public void pop() { v=null; }

        private Object v = null;
}
```

Note that it is crucial to specify the *LIFO property* of Stacks.

NonStack is not a subtype of *Stack* since there is an *observable change in behavior*.

Is there an algorithm that can determine, in general, whether one type *S* is a subtype of another type *T* ? The answer is NO. This is similar to asking the question "Suppose we have a program, *P*, that uses a type *T*. Furthermore suppose we modify *P* to yield *P'* where we replace (some or all) occurrences of *T* in *P* with *S*. Will we obtain the same results in all cases whether we run *P* or *P'* ?"

It is impossible, in general, to determine if two programs exhibit the same behavior (the argument is based on the ***Halting Problem)***. Therefore, it is impossible to determine if

one type is a subtype of another type. The solution language designers use to support the *Principle of Substitution* (subtyping) is to provide various syntactic and semantic requirements in the language. For example, in *Java* we can use subclasses as subtypes to allow substitutability. There are other language features to help support this viewpoint. Clearly this does not always work – see the *NonStack* example above.

The designers of Java took the viewpoint that classes are types and subclasses are subtypes. Therefore, the *Principle of Substitutability* applies so we can use e.g. IdTree wherever we expect AST – this is alright since there is no change in behavior.

Forms of Inheritance

1. **Specialization**: class B extends A

   B is-a ***special case*** of an A class Student extends Person

   class B exhibits all of the behaviour of class A. In addition, *B overrides at least one method in A to specialize it for the needs of B objects.*

2. **Specification**: specify an interface; defer method bodies

   Abstract classes and interfaces

   e.g. public abstract class AST {

   ...

   public **abstract** Object accept(ASTVisitor v); // deferred body

   }

```
public interface Iterator<E> {
  boolean hasNext();
  E next();
  void remove();
}
```

3. **Construction**: re-use the code found in the Parent class (of questionable value)

```
class Stack extends Vector {
    public Object push(Object item) {
            addelement(item); return item; }
    public boolean empty() {
            return isEmpty(); }
...
}
```

Note: *Construction* breaks the *Principle of Substitutability*. We cannot freely substitute *Stacks* for *Vectors* without any observable change in behavior. Note that in some object oriented languages we can attempt to enforce the *Principle of Substitutability* by using pre- and post-conditions on instance interactions. This would preclude the use of any Vector methods that do not adhere to LIFO. Therefore, Stacks cannot be used wherever a Vectors is used.

4. **Generalization/ Extension**: we build on a base of existing classes that we do not wish to, or cannot, modify – ***expands on base, does not change or specialize base.***. *Extension* includes functionality that is less strongly tied to the existing methods of the parent (note in the example below that there is new functionality that is not meaningful to the parent)

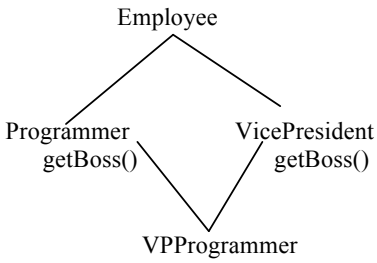
> public class Properties extends Hashtable
>
> The Properties class represents a persistent set of properties. The Properties can be saved to a stream or loaded from a stream. Each key and its corresponding value in the property list is a string.
>
> public Properties() -- creates an empty property list with no default values.
>
> public String getProperty(String key)
> Searches for the property with the specified key in this property list.
>
> public void list(PrintStream out)
> Prints this property list out to the specified output stream.
>
> public synchronized void load(InputStream in) throws IOException
> Reads a property list from an input stream.
>
> public Enumeration propertyNames()
> Returns an enumeration of all the keys in this property list, including the keys in the default property list.

5. **Limitation**: can't modify parent (e.g parent class is in the Java library which can't be modified) so we override the non-relevant methods! Did we override ALL non-relevant methods from ALL the superclasses??! The behavior is more limited in the child then in the parent.

```
class Set extends Vector {
      // we can re-use addElement, etc. from Vector

      public elementAt (int index) { return null; }
      // override this parent method
}
```

Clearly, limitation violates the *Principle of Substitutability* – e.g. the *elemantAt* behavior is *different in the subclass from that in the base class*.

6. **Combination:** multiple inheritance; many thorny practical issues; it's not allowed in Java



VPProgrammer vpp = new VPProgrammer()

vpp.getBoss() -- which *getBoss*() method do we use??

Modifiers

public -- accessible to all

private -- not accessible outside of current class

protected -- accessible to subclasses

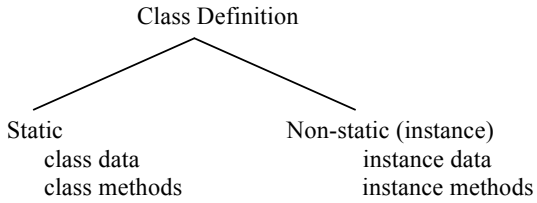
<none> -- accessible to other classes in the same package

final -- can't subclass, can't override

static -- relates to the CLASS not the INSTANCE

abstract -- body of the method is deferred; class definition is incomplete

- can't instantiate



e.g.

```
public class Sym {
  public static final int Program = 0;
  public static final int Int = 1;
  public static final int BOOLean = 2;


public final class String extends Object implements Serializable {
      public static String valueOf(int i)
          Returns the string representation of the int argument.
      ...
}
```

Benefits of Inheritance

1. **Reuse code/data** from parent
2. **Increased reliability** - when code is used in 2 or more applications it's used frequently so there is a greater probability that bugs will be exposed
3. **Code sharing -** use classes in several applications; software IC's; Java library (Vectors, etc.)
4. **Consistency of interface** - Subclasses inherit SAME behavior from parent
5. **Rapid prototyping** - use lots of existing components to more rapidly build new components
6. **Frameworks** - frameworks rely on inheritance and substitutability to provide large amounts of code to be reused for many applications (e.g. AWT, MFC)

Cost of Inheritance

1. Execution speed - slower with respect to *dynamic binding*

```
public class ProgramTree extends AST {
   public ProgramTree() {}

   public Object accept(ASTVisitor v) { return v.visitProgramTree(this);  }

}
```

Which *visitProgramTree ()* method body should we execute at runtime? -- we don't know until *runtime* which ***concrete subclass*** of ASTVisitor **i**s referenced by *v*. Therefore, we must *execute runtime code* to examine the class of the object referred to by *v* to determine which *visitProgramTree* method body to associate (bind to) with the *visitProgramTree* reference in *v.visitProgramTree(this)*; this is runtime or *dynamic information - dynamic binding*

We are using dynamic binding, which adds to the program runtime

Note that dynamic binding is an essential (extremely powerful) aspect of OOP

2. **Program size** - use large libraries such as the Java libraries (may not actually need many parts of the libraries); but memory costs are quickly decreasing

3. **Message passing overhead** - more costly then using simple procedures; but, this OOP paradigm greatly increases programmer productivity so it's well worth it

4. **Program complexity** - tight coupling within inheritance hierarchy; changes higher up might have dramatic changes below; research in refactoring code

Reading:

Budd text: Chapter 8

XII. The Interpreter

Frames (Activation Records) and the Runtime stack

The set of variables (actual arguments, locals, temps) for each function are stored in the ***activation record*** or ***frame***..

Since functions are called/returned from in a ***LIFO*** fashion we use a **stack** to hold the frames - the ***runtime stack***

If we have the calling sequence:

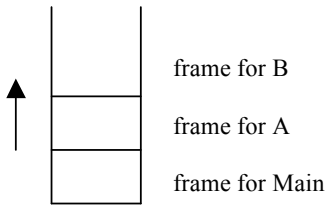
Main program

...

call A (called from Main)

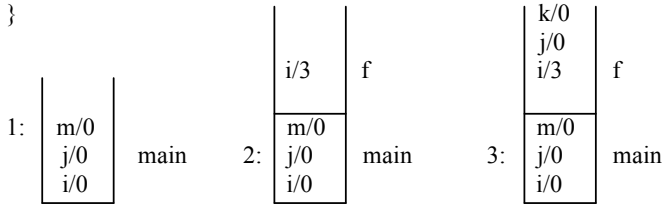
call B (called from A)

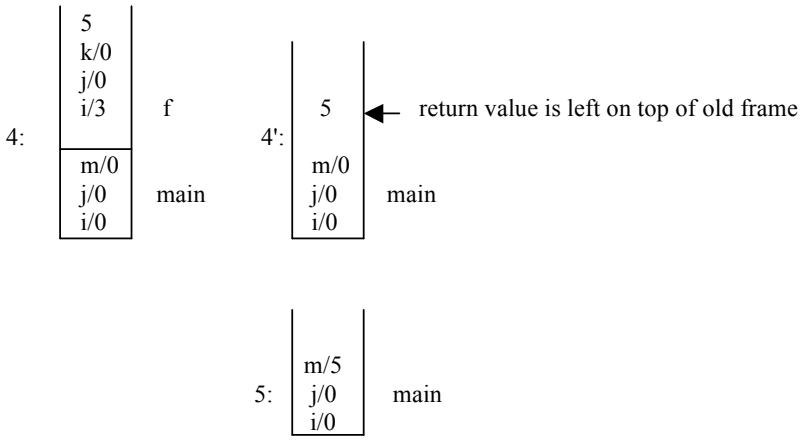
Runtime Stack:



Frames: arguments + local variables + temporary storage

```
program { int i       int j
    int f(int i)     {          (2)
            int j          int k          (3)
            return i + j + k  + 2  (4) the sum will be on the stack just prior to return
    }
int m            (1)
m = f(3)         (5)
i = write(j+m)
}
```





Summary of the X-machine Bytecodes

Bytecode	Description	Examples
HALT	*halt* execution	HALT
POP	*POP n*: Pop top *n* levels of runtime stack	POP 5 POP 0
FALSEBRANCH	*FALSEBRANCH <label>* - pop the top of the stack; if it's *false* (0) then branch to <label> else execute the next bytecode	FALSEBRANCH xyz<<3>>
GOTO	*GOTO <label>*	GOTO xyz<<3>>
STORE	*STORE n <id>* - pop the top of the stack; store value into the *offset n from the start of the frame*; <id> is used as a comment, it's the variable name where the data is stored	STORE 2 i
LOAD	*LOAD n <id>*; push the value in the slot which is *offset n from the start of the frame* onto the top of the stack; <id> is used as a comment, it's the variable name from which the data is loaded	LOAD 3 j
LIT	*LIT n* - load the *literal value n* *LIT 0 i* – this form of the Lit was generated to load 0 on the stack in order to initialize the variable *i* to 0 and reserve space on the runtime stack for *i*	LIT 5 LIT 0 i
ARGS	*ARGS n* ;Used prior to calling a function: n = #args this instruction is *immediately followed* by the *CALL* instruction; the function has *n args* so *ARGS n* instructs the interpreter to set up a new frame *n down from the top*, so it will include the arguments	ARGS 4
CALL	*CALL <funcname>* - transfer control to the indicated function	CALL f CALL f<<3>>
RETURN	*RETURN <funcname>* ; Return from the current function; *<funcname>* is used as a comment to indicate the current function *RETURN* is generated for intrinsic functions	RETURN f<<2>> RETURN
BOP	*bop <binary op>* - *pop top 2 levels* of the stack and perform indicated operation – operations are + - / * == != <= > >= < | & *| and & are logical operators, not bit operators* lower level is the *first operand*: e.g. *<second-level> + <top-level>*	BOP +
READ	*READ* ; Read an integer; prompt the user for input; put the value just read on top of the stack	READ
WRITE	*WRITE* ; Write the value on top of the stack to output; *leave the value on top of the stack*	WRITE
LABEL	*LABEL <label>;* target for branches; (see *FALSEBRANCH, GOTO*)	LABEL xyz<<3>> LABEL Read

Bytecodes	***Source Program***
GOTO start<<1>>	program {
LABEL Read	
READ	
RETURN	
LABEL Write	
LOAD 0 dummyFormal	<bodies for read/write functions>
WRITE	
RETURN	
LABEL start<<1>>	
LIT 0 i	int i
LIT 0 j	int j
GOTO continue<<3>>	
LABEL f<<2>>	int f(int i) {
LIT 0 j	int j
LIT 0 k	int k
LOAD 0 i	i + j + k + 2
LOAD 1 j	
BOP +	
LOAD 2 k	
BOP +	
LIT 2	
BOP +	
RETURN f<<2>>	return i + j + k + 2
POP 2	<remove local variables - j,k>
LIT 0 GRATIS-RETURN-VALUE	
RETURN f<<2>>	
LABEL continue<<3>>	
LIT 0 m	int m
LIT 3	f(3)
ARGS 1	
CALL f<<2>>	
STORE 2 m	m = f(3)
LOAD 1 j	j + m
LOAD 2 m	
BOP +	
ARGS 1	
CALL Write	write(j+m)
STORE 0 i	i = write(j+m)
POP 3	<remove local variables - i,j,m>
HALT	

Execution Trace

Bytecodes executed	***Runtime Stack after Executing Bytecode***
	<stack grows to the right>
GOTO start<<1>>	--
LABEL start<<1>>	
LIT 0	[0]
LIT 0	[0,0]
GOTO continue<<3>>	[0,0]
LABEL continue<<3>>	[0,0]
LIT 0	[0,0,0]
LIT 3	[0,0,0,3]
ARGS 1	[0,0,0] [3] -- one arg loaded on top of stack
CALL f<<2>>	[0,0,0] [3] -- new frame for *f*
LABEL f<<2>>	[0,0,0] [3]
LIT 0	[0,0,0] [3,0]
LIT 0	[0,0,0] [3,0,0]
LOAD 0	[0,0,0] [3,0,0,3]
	-- load from offset 0 in current frame
LOAD 1	[0,0,0] [3,0,0,3,0]
BOP +	[0,0,0] [3,0,0,3]
	-- add top 2 levels; replace with sum
LOAD 2	[0,0,0] [3,0,0,3,0]
BOP +	[0,0,0] [3,0,0,3]
LIT 2	[0,0,0] [3,0,0,3,2]
BOP +	[0,0,0] [3,0,0,5]
RETURN	[0,0,0,5]
	-- move top to start of frame; reset frame
STORE 2	[0,0,5]

Stack Changes:	+1	-1	0
	LIT	BOP	GOTO
	LOAD	STORE	LABEL

Simple Interpreter Schema

1. Load the bytecodes generated by the compiler
2. Execute the codes in the Virtual Machine

class *ByteCode* is the *abstract class* which each bytecode (in-) directly extends

e.g. *class ReadCode extends ByteCode*

The ByteCodeLoader (BCL)

- Reads in the next bytecode
- Builds an instance of the class corresponding to the bytecode - e.g. if we read *LIT 2*

  then we build a new *LitCode* instance
- The bytecode class instance is added to the ***Program***
- After all bytecodes are loaded, the symbolic addresses are resolved - e.g.

```
 * The Program class will hold the bytecode program loaded from file
 * It will also resolve symbolic addresses in the program - e.g.
 * if we have the following bytecode program
 *
 * 0. FALSEBRANCH continue<<6>>
 * 1. LIT 2
 * 2. LIT 2
 * 3. BOP ==
 * 4. FALSEBRANCH continue<<9>>
 * 5. LIT 1
 * 6. ARGS 1
 * 7. CALL Write
 * 8. STORE 0 i
 * 9. LABEL continue<<9>>
 * 10. LABEL continue<<6>>
 * <br>
 * The addresses in the program above will be resolved to:<br>
 *
 * 0. FALSEBRANCH 10
 * 1. LIT 2
 * 2. LIT 2
 * 3. BOP ==
 * 4. FALSEBRANCH 9
 * 5. LIT 1
 * 6. ARGS 1
 * 7. CALL <address of Write>
 * 8. STORE 0 i
 * 9. LABEL continue<<9>>
 * 10. LABEL continue<<6>>
```

e.g.

File	**CodeTable**		**Program**
	Keys	***Values***	
LIT 1	"LIT"	"LitCode"	<LitCode instance>
LOAD 2	"LOAD"	"LoadCode"	<LoadCode instance>
READ	"HALT"	"HaltCode"	<ReadCode instance>
	...	...	
	Strings	*Strings*	

CodeTable

- Used by BCL
- Contains a *HashMap* with keys being the bytecode strings (e.g. "HALT","POP")

  and values being the name of the corresponding class (e.g. "HaltCode","PopCode")

```
 *  ByteCodeLoader class will load the bytecodes from the file into the Virtual Machine
 *  When the bytecodes are loaded we'll get the string for the bytecode and then we'll get
  * the bytecode class for that string from the
 *  Codetable to construct an instance of the bytecode; e.g.
 *  if the bytecode loader reads the line: <br>
 *  HALT  <br>
 *  then we'll build an instance of the class HaltCode and store that instance in the
 *  Program object;
```

The code table can be populated through an initialization method. It is ok to hard-code the statements that populate the data in the *CodeTable* class.

The following statements are NOT part of the CodeTable class – they are included here

for illustrative purposes

```
String code = "HALT";

String codeClass = CodeTable.get(code);
ByteCode bytecode =
   (ByteCode)(Class.forName("interpreter."+codeClass).newInstance());
```

// Note that the *newInstance* method requires an *accessible* no-argument constructor for

// the class; by *accessible constructor* we mean that the constructor should be e.g. ***public***

Class.forName("interpreter.HaltCode") will return the class *interpreter.HaltCode*

(Class.forName("interpreter.HaltCode").newInstance()); will build a *new instance* of

the class *interpreter.HaltCode*

Note the ability to *dynamically create instances* (we don't know which classes will be

used *statically*). **This dynamic nature is extremely important** in the context of

e-commerce with dynamically produced content/dynamic web pages/plugins added to

e.g. Netscape ==> just ***provide an interface*** for the plugins and they are useable.

Check http://unixlab.sfsu.edu/~levine/common/csc413/JavaTechTips/ for more on reflection

Javadoc Documentation of Selected Interpreter Classes

The documentation is obtained by changing directory to the root directory of the system:
d:cd d:\
d:javadoc interpreter -- this creates the documentation for the interpreter package

Class interpreter.Program

```
java.lang.Object
   |
   +----interpreter.Program
```

public class **Program**

extends Object

```
  The Program class will hold the bytecode program loaded from
file
  It will also resolve symbolic addresses in the program - e.g.
  if we have the following bytecode program:
  1. FALSEBRANCH continue<<6>>
  2. LIT 2
  3. LIT 2
  4. BOP ==
  5. FALSEBRANCH continue<<9>>
  6. LIT 1
  7. ARGS 1
  8. CALL Write
  9. STORE 0 i
  10. LABEL continue<<9>>
  11. LABEL continue<<6>>

  The addresses in the program above will be resolved to:
  1. FALSEBRANCH 11
  2. LIT 2
  3. LIT 2
  4. BOP ==
  5. FALSEBRANCH 10
  6. LIT 1
  7. ARGS 1
  8. CALL <address of Write>
  9. STORE 0 i
  10. LABEL continue<<9>>
  11. LABEL continue<<6>>
```

Class interpreter.CodeTable

```
java.lang.Object
   |
   +----interpreter.CodeTable
```

public class **CodeTable**

extends Object

get

```
public static String get(String code)
```

init

```
public static void init()
```

init each entry with the name of the corresponding class for the code; when the bytecodes are loaded we'll get the string for the bytecode and then we'll get the bytecode class for that string from the Codetable to construct an instance of the bytecode; e.g. if the bytecode loader reads the line:
HALT
then we'll build an instance of the class HaltCode and store that instance in the Program object; If we add more bytecodes then we'll need another class for the bytecode to indicate how to interpret it and we'll have to add the code to the table below

Class interpreter.ByteCodeLoader

```
java.lang.Object
   |
   +----interpreter.ByteCodeLoader
```

public class **ByteCodeLoader**

extends *Object*

ByteCodeLoader class will load the bytecodes from the file into the Virtual Machine
When the bytecodes are loaded we'll get the string for the bytecode and then we'll get the bytecode class for that string from the Codetable to construct an instance of the bytecode; e.g. if the bytecode loader reads the line:
HALT
then we'll build an instance of the class HaltCode and store that instance in the Program object;

ByteCodeLoader

```
public ByteCodeLoader(String programFile) throws IOException
```

loadCodes

Load all the codes from file; request Program to resolve any branch addresses

from symbols to their address in code memory, e.g.

```
  1. GOTO addr            GOTO 5
  2. LOAD 3               LOAD 3
  3. LIT 2       ==>      LIT 2
  4. STORE 4              STORE 4
  5. LABEL addr           LABEL addr

Return the bytecode program in an appropriate data structure for
processing by the VirtualMachine

Note that just prior to returning, this method will request the
constructed Program to resolve all addresses
```

The Runtime Stack

Records and processes the stack of active frames

Uses:

Stack framePointers -- used to record *prior frame pointers* when calling

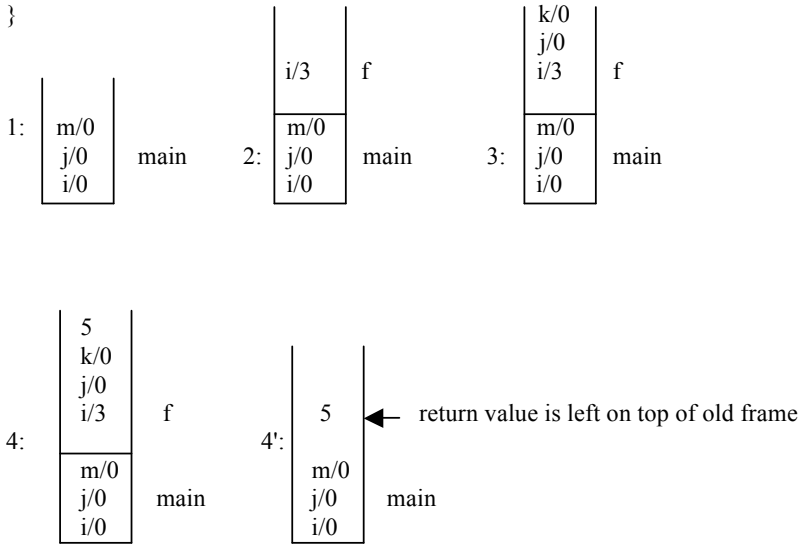
-- functions

Vector runStack -- the runtime stack; it's a *Vector* rather than a *Stack*

-- since we need access to ALL locations in the current

-- frame, not just the top location

Recall the simulation with the runtime stack presented earlier:



Initially the *framePointers* will have 0 for step 1 above; when we call the function *f* at step 2 *framePointers* will have 03 since 0 is the start of the main frame and 3 is the start of the frame for *f;* At step 4' we pop the *framePointers stack (3 is popped)*;

framePointers will only have 0 on the top

Class interpreter.RunTimeStack

```
java.lang.Object
   |
   +----interpreter.RunTimeStack
```

public class **RunTimeStack**

The RunTimeStack class maintains the stack of active frames; when we call a function we'll push a new frame on the stack; when we return from a function we'll pop the top frame

RunTimeStack

```
public RunTimeStack()
```

dump

```
public void dump()
```

Dump the RunTimeStack information for debugging

peek

```
public int peek()
```

Returns:

the top item on the runtime stack

pop

```
public int pop()
```

pop the top item from the runtime stack

Returns:

that item

push

```
public int push(int i)
```

Parameters:

i - push this item on the runtime stack

Returns:

the item just pushed

newFrameAt

```
public void newFrameAt(int offset)
```

start new frame

Parameters:

offset - indicates the number of slots down from the top of RunTimeStack for starting the new frame

popFrame

```
public void popFrame()
```

We pop the top frame when we return from a function; before popping, the function's return value is at the top of the stack so we'll save the value, pop the top frame and then push the return value

store

```
public int store(int offset)
```

Used to store into variables

load

```
public int load(int offset)
```

Used to load variables onto the stack

push

```
public Integer push(Integer i)
```

Used to load literals onto the stack - e.g. for lit 5 we'll call push with 5

The Virtual Machine

RunTimeStack runStack

int pc	-- the program counter
Stack returnAddrs	-- push/pop when call/return functions
boolean isRunning	-- true while the VM is running
Program program	-- bytecode program

```
public void executeProgram() {
    pc = 0;
    runStack = new RunTimeStack();
    returnAddrs = new Stack();
    isRunning = true;
    while (isRunning) {
        ByteCode code = program.getCode(pc);
        code.execute(this);
        // runStack.dump();    // check that the operation is correct
        pc++;
    }
}
```

Note that we can ***easily add new bytecodes*** without affecting the operation of the Virtual Machine We are using ***dynamic binding*** to achieve code flexibility, extendability and readability (note how easy it is to read and understand this code).

The *returnAddrs* stack stores the bytecode index (PC) that the virtual machine should execute when the current function exits. Each time a function is entered, the PC should be pushed on to the *returnAddrs* stack. When a function exits the PC should be restored to the value that is popped from the top of the *returnAddrs* stack.

Interpreter.java

```
package interpreter;

import java.io.*;

/**
 * <pre>
 *
 *
 *
 *    Interpreter class runs the interpreter:
 *    1. Perform all initializations
 *    2. Load the bytecodes from file
 *    3. Run the virtual machine
 *
 *
 *
 * </pre>
 */
public class Interpreter {

        ByteCodeLoader bcl;

        public Interpreter(String codeFile) {
                try {
                        CodeTable.init();
                        bcl = new ByteCodeLoader(codeFile);
                } catch (IOException e) {
                        System.out.println("**** " + e);
                }
        }

        void run() {
                Program program = bcl.loadCodes();
                VirtualMachine vm = new VirtualMachine(program);
                vm.executeProgram();
        }

        public static void main(String args[]) {
                if (args.length == 0) {
                        System.out.println("***Incorrect usage, try: java interpreter.Interpreter <file>");
                        System.exit(1);
                }
                (new Interpreter(args[0])).run();
        }
}
```

Coding Hints - Suggested Order to Write Code

1. Bytecode stubs - just include empty method bodies
2. CodeTable
3. Bytecode Loader
4. Program (code resolveAddress method)
5. RunTimeStack
6. Interpreter (given here)
7. Virtual Machine with dump method
8. Fill in Bytecode stubs

Additional Requirements

1. You should be able to execute your program by typing:
Java interpreter <bytecode file>
e.g.,
java interpreter factorial.x.cod

You will need to create a jar file named interpreter.jar to submit via email to the grader (see Appendix A).
You can assume that the bytecode programs you use for testing are generated correctly, and therefore should not contain any errors. You *should check* for stack overflow/underflow errors, or popping past a frame boundary.

2. You should provide as much documentation as is necessary to clearly explain your code. Short, OBVIOUS methods don't need comments, however you should comment every class to describe its function. Take into consideration that the first level of documentation is your code - it should be clear with appropriately named variables, etc. If you need to elaborate on algorithms that are not apparent, then include Javadoc comments for those sections of code.

It is also a good idea to follow the standard Java coding conventions that are recommended by Sun. Code conventions improve the readability of the software, allowing engineers to understand new code more quickly and thoroughly.

http://www.oracle.com/technetwork/java/codeconv-138413.html

3. DO NOT provide a method that returns components contained WITHIN the VM (this is the exact situation that will break encapsulation) - you should request the VM to perform operations on its components. This implies that the VM owns the components and is free to change them, as needed, without breaking clients' code (e.g., suppose I decide to change the name of the variable that holds my runtime stack - if your code had referenced that variable then your code would break. This is not an unusual situation - you can consider the names of methods in the Java libraries that have been deprecated).

The only downside is it might be a bit inefficient. Since I want to impress on everyone important software engineering issues, such as encapsulation benefits, I want to enforce the requirement that you do not break encapsulation. Consider that the VM calls the individual ByteCodes' *execute* method and passes itself as a parameter. For the ByteCode to execute, it must invoke 1 or more methods in the runStack. It can do this by executing *VM.runStack.pop()*; however, this does break encapsulation. To avoid this, you'll need to have a corresponding set of methods within the VM that do nothing more than pass the call to the runStack. e.g., you would want to define a VM method

public int popRunStack(){return runStack.pop();}

called by, e.g.,

int temp = VM.popRunStack();

4. Each bytecode class should have fields for its specific arguments. The abstract *ByteCode* class SHOULD NOT CONTAIN ANY FIELDS (instance variables) THAT RELATE TO ARGUMENTS. *This is a design requirement*.

It's easier to think in more general terms (i.e. plan for any number of args for a bytecode). Note that the Bytecode abstract class should not be aware of the peculiarities of any particular bytecode. That is, some bytecodes might have zero args (HALT), or one arg, etc. Consider providing an"init" method with each bytecode class. After constructing the bytecode (e.g. LoadCode) then you can call its"init" method and give it a vector of String args. Each bytecode object will interrogate the vector and extract relevant args for itself. The Bytecode class SHOULD NOT record the args for any bytecodes. Each concrete bytecode class will have instance variable(s) to record its args.

When you read a line from the bytecode file, you should parse each argument placing them into a vector. Each bytecode takes responsibility for extracting relevant information from the vector and storing it as private data.

5. The Bytecodes should be contained in a subpackage of the interpreter package (*interpreter.bytecode*)

6. Any output produced by the WRITE bytecode will be interspersed with the output from dumping (if dumping is turned on). In the Write action you should only print one number PER LINE. DO NOT output something like:

program output: 2

Only output the actual number on the line by itself.

7. There is no need to check for a divide-by-zero error. Assume that you will not have a test case where this occurs.

DUMPING PROGRAM STATE

You should include 1 additional bytecode: DUMP
The formats are:
DUMP ON
and
DUMP OFF

DUMP ON is an interpreter command to turn on runtime dumping. This will set an interpreter switch that will cause dumping ***AFTER*** execution of each bytecode.

DUMP OFF will reset the switch to end dumping.
Note that the DUMP instructions will not be printed.

DO NOT dump program state unless dumping is turned on

Consider the following bytecode program:

```
GOTO start<<1>>
LABEL Read
READ
RETURN
LABEL Write
LOAD 0 dummyFormal
WRITE
RETURN
LABEL start<<1>>
LIT 0 i
LIT 0 j
GOTO continue<<3>>
LABEL f<<2>>
LIT 0 j
LIT 0 k
LOAD 0 i
LOAD 1 j
DUMP OFF
BOP +
LOAD 2 k
BOP +
LIT 2
BOP +
RETURN f<<2>>
POP 2
LIT 0    GRATIS-RETURN-VALUE
RETURN f<<2>>
LABEL continue<<3>>
DUMP ON
LIT 0 m
LIT 3
ARGS 1
CALL f<<2>>
DUMP ON
STORE 2 m
DUMP OFF
```

```
LOAD 1 j
LOAD 2 m
BOP +
ARGS 1
CALL Write
STORE 0 i
POP 3
HALT
```

When dumping is turned on you should print the following information **JUST AFTER** executing the next bytecode

- ·print the bytecode that was just executed (DO NOT PRINT the DUMP bytecodes)
- ·print the runtime stack *with spaces separating frames* (just after the bytecode was executed)
- **if dump is not on then do not print the bytecode, nor dump the runtime stack**

Following is an example of the expected printout from the program given above (we only give a portion of the printout as an illustrative example):

```
LIT 0 m     int m
[0,0,0]
LIT 3
[0,0,0,3]
ARGS 1
[0,0,0] [3]
CALL f<<2>>     f(3)
[0,0,0] [3]
LABEL f<<2>>
[0,0,0] [3]
LIT 0 j     int j
[0,0,0] [3,0]
LIT 0 k     int k
[0,0,0] [3,0,0]
LOAD 0 i     <load i>
[0,0,0] [3,0,0,3]
LOAD 1 j     <load j>
[0,0,0] [3,0,0,3,0]
...
STORE 2 m     m = 5
[0,0,5]
```

Note:
Following shows the output if dump is on and 0 is at the top of the runtime stack
RETURN f<<2>>   exit f: 0   note that 0 is returned from f<<2>>

Notes:
If Dumping is turned on and we encounter an instruction such as *WRITE* then *output as usual*; the value may be printed either before or after the dumping information e.g.,
LIT 3
0003
3
WRITE
0003
The following dumping actions are taken for the indicated bytecodes, other bytecodes do not have special treatment when being dumped (see example above):

LIT 0 <id> int <id> note: for simplicity ALWAYS assume this *lit* is an *int* declaration
  e.g. LIT 0 j int j
LOAD c <id> <load <id>>
  e.g. LOAD 2 a <load a>
STORE c <id> <id> = <top-of-stack>
  e.g. if the stack has:
  0123
  and we execute STORE 1 k
  then we will dump
  STORE 1 k k = 3
RETURN <id> exit <base-id>: <value>
  <value> is the value being returned from the function
  <base-id> is the actual id of the function; e.g. for *RETURN f<<2>>*, the <base-id> is *f*
CALL <id> <base-id>(<args>)
  e.g. if the stack has:
  0123
  and we execute CALL f<<3>> after executing ARGS 2 then we will dump
  CALL f<<3>> f(2,3)
    we strip out any brackets (<, >); the ARGS bytecode just seen tells us that we have a function with 2 args, which are on the top 2 levels of the stack - the first arg was pushed first, etc.

Dumping Implementation Notes

1. The virtual machine maintains the state of your running program, so it is a good place to have the *dump* flag. You should not use a static variable in the *ByteCode* class.

The *dump* method should be part of the *RunTimeStack* class. This method is called without any arguments. Therefore, there is no way to pass any information about the *VM* or the *ByteCode* Classes into *RunTimeStack*. As a result, you can't really do much dumping inside *RunTimeStack.dump()* except for dumping the contents of *RunTimeStack* itself.

2. It is impossible to determine the declared type of a variable by looking at the bytecode file. To simplify matters you should assume whenever the interpreter encounters LIT 0 x (for example), that it is an int. When you are dumping the bytecodes it is ok to represent the value as an int.

Be sure to use Javadoc comments within your code (as is done in the code found herein); you will not turn in the result of running Javadoc since the source programs will be graded.

Reading:

Core Java 2: Chapter 12

XIII. The Debugger

Following is a sample of debugger commands. YOU WILL ONLY IMPLEMENT THE COMMANDS SPECIFIED IN THE ASSIGNMENT FOUND ON ***ilearn***, NOT ALL OF THE FOLLOWING COMMANDS! Create a debugger for the Virtual Machine with facilities drawn from the following (see assignment specs for required commands to implement):

Step over a line (actually, to the next line containing one of the constructs listed in point 3 below)
Step into a function call on the current line; if there is no function call then just ***step over***
Step out *of current activation* of a function call
Set breakpoints (valid locations for setting breakpoints are mentioned in the notes, below)
List current breakpoint settings
Clear designated breakpoints
Set function tracing - print function name (indented according to function nesting) argument values at entry/result at exit
Print Call Stack – refer to *Netbeans* printing of call stack in the *debug* mode (print stack of function calls, along with line numbers of calling function)
Display local variables
Change values of local variables
Watch local variables - if you set a watch on a local variable, *j*, then whenever *j*'s value changes you should notify the user of the new value; when *j* goes out of scope then the watch is removed; if *j* is *shadowed* by a declaration of *another j* in an inner block then the watch is not removed
Display the source code of the current function with an indication where execution has stopped
Continue execution from the current point
Re-execute the current function – unravel execution, back to when the current function started executing and stop there; the intent is to examine the current function more closely to find errors therein
Evaluate expression – use exercise 3 at the end of this Reader to evaluate simple expressions in the debugger
Halt execution

Debugger Notes

Use an interpreter debug switch to check for debug bytecodes and provide the debugging operations (eg. *java -jar interpreter.jar -d factorial*). Just after the Interpreter starts executing (when the VM is about ready to execute the first instruction) you should allow the user to access all of the debugger commands to set breakpoints, etc. Be sure to provide a *help menu* in a reasonable format.

Breakpoints may only be set at the beginning of the following constructs (note these constructs will have an associated LINE bytecode):

blocks decl if while return assign

e.g. for the following source program:

```
1.  program {boolean j int i
2.    int factorial(int n) {
3.       if (n < 2) then       <== The if node will have the boundaries of 3 and 6 to indicate that
```

```
4.          { return 1 }        <== the if construct starts at line 3 and ends at line 6.
5.        else                  <== a breakpoint cannot be set on this line
6.          {return n*factorial(n-1) }
7.    }                         <== a breakpoint cannot be set on this line
8.    while (1==1) {
9.        i = write(factorial(read()))
10.   }
11.  }
```

We will need new Bytecodes:

LINE n

n is the *current* source line number; the generated bytecodes for line n follow this *LINE n* code

FUNCTION name start end
FORMAL xyz offset

The FUNCTION and FORMAL bytecodes will be generated as a header for each function declaration. The bytecodes for each function will begin with:

```
LABEL name1                     -- branch label for function call
LINE n                          -- start of function definition
FUNCTION name start end         -- name of function with source line
                                -- number boundaries given by start
                                -- and end
FORMAL f1 0                     -- f1 is first formal with offset 0
FORMAL f2 1                     -- f2 is second formal
```

Be sure to provide a user interface of professional quality (correct English, user-friendly, etc.)

Debugger Structures - Interpreter Modifications (hints)

A ***Vector*** of *entries* where each entry contains

String sourceLine -- contains the source program line i for Vector slot i
Boolean isBreakptSet -- is a breakpoint set for this line?

A Stack (environmentStack) -- do not create a new class for this structure – it is a Stack
-- each entry contains information for each frame (a FunctionEnvironmentRecord

Push on function entry
Pop on function exit
Each stack entry (FunctionEnvironmentRecord) contains:

a Symbol Table similar to the one used by the constrainer; you should copy the code from the constrainer's Symbol Table and use it, with appropriate modifications. DO NOT CREATE YOUR OWN SYMBOL TABLE MECHANISM.
The table is modified as follows when the given bytecodes are encountered:

FORMAL xyz n	enter("xyz",n) -- enter into Symbol Table
LIT 0 i	enter("i",current offset)
POP n	delete the last n items entered into the Symbol Table
int startLine	starting source line number for function
int endLine	ending source line number for function
int currentLine	maintains the current line number being processed by the VM; note that this must be re-set when branching instructions are processed.
String name	Name of current function

Use of Environment Stack with New Bytecodes

```
1.  program {boolean j int i
2.    int factorial(int n) {
3.       if (n < 2) then
4.          { return 1 }
5.       else
6.          {return n*factorial(n-1) }
7.    }
8.    while (1==1) {
9.       i = write(factorial(read()))
10.   }
11.  }
```

Example of Generated Bytecodes for the Factorial Program Given Above

```
GOTO start<<1>>
LABEL Read
LINE -1                    <== the following codes are for the intrinsic function so we
FUNCTION Read -1 -1            use line numbers that are outside of the program bounds
READ
RETURN
LABEL Write
LINE -1
FUNCTION Write -1 -1
FORMAL dummyFormal 0
LOAD 0 dummyFormal
WRITE
RETURN
LABEL start<<1>>
LINE 1
FUNCTION main 1 11
LIT 0 j
LIT 0 i
GOTO continue<<3>>
LABEL factorial<<2>>
LINE 2
FUNCTION factorial 2 7
FORMAL n 0
LINE 3
LOAD 0 n
LIT 2
BOP <
FALSEBRANCH else<<4>>
LINE 4
```

```
LIT 1
RETURN factorial<<2>>
POP 0
GOTO continue<<5>>
LABEL else<<4>>
LINE 6
LOAD 0 n
LOAD 0 n
LIT 1
BOP -
ARGS 1
CALL factorial<<2>>
BOP *
RETURN factorial<<2>>
POP 0
LABEL continue<<5>>
POP 0
LIT 0    GRATIS-RETURN-VALUE
RETURN factorial<<2>>
LABEL continue<<3>>
LABEL while<<7>>
LINE 8
LIT 1
LIT 1
BOP ==
FALSEBRANCH continue<<6>>
LINE 9
ARGS 0
CALL Read
ARGS 1
CALL factorial<<2>>
ARGS 1
CALL Write
STORE 1 i
POP 0
GOTO while<<7>>
LABEL continue<<6>>
POP 2
HALT
```

Simulation of Debugger Execution for the Factorial Program

Bytecode	**Environment Stack** (symtab,start,end,name,currentLine)
<start>	<-,-,-,-,->
GOTO start<<1>>	
LABEL start<<1>>	
LINE 1	<-,-,-,-,1>
FUNCTION main 1 11	<(),1,11,main,1>
LIT 0 j	< (<j,0>),1,11,main,1>
LIT 0 i	< (<j,0>,<i,1>),1,11,main,1>
GOTO continue<<3>>	
LABEL continue<<3>>	
LABEL while<<7>>	
LINE 8	< (<j,0>,<i,1>),1,11,main,8>
LIT 1	
LIT 1	
BOP ==	
FALSEBRANCH continue<<6>>	
LINE 9	< (<j,0>,<i,1>),1,11,main,9>
ARGS 0	
CALL Read	< (<j,0>,<i,1>),1,11,main,9><(),-,-,-,->
LABEL Read	
LINE -1	< (<j,0>,<i,1>),1,11,main,9><(),-,-,-,-1>
FUNCTION Read -1 -1	< (<j,0>,<i,1>),1,11,main,9><(),-1,-1,Read,-1>
READ	
RETURN	< (<j,0>,<i,1>),1,11,main,9> *return to line 9*
ARGS 1	
CALL factorial<<2>>	< (<j,0>,<i,1>),1,11,main,9><(),-,-,-,->
LABEL factorial<<2>>	
LINE 2	< (<j,0>,<i,1>),1,11,main,9><(),-,-,-,2>
FUNCTION factorial 2 7	< (<j,0>,<i,1>),1,11,main,9><(),2,7,factorial,2>
FORMAL n 0	< (<j,0>,<i,1>),1,11,main,9><(<n,0>),2,7,factorial,2>
LINE 3	< (<j,0>,<i,1>),1,11,main,9><(<n,0>),2,7,factorial,3>
LOAD 0 n	
LIT 2	
BOP <	
FALSEBRANCH else<<4>>	
LABEL else<<4>>	
LINE 6	< (<j,0>,<i,1>),1,11,main,9><(<n,0>),2,7,factorial,6>
LOAD 0 n	
LOAD 0 n	
LIT 1	
BOP -	
ARGS 1	
CALL factorial<<2>>	

<(<j,0>,<i,1>),1,11,main,9><(<n,0>),2,7,factorial,6><(),-,-,-,->
LABEL factorial<<2>>
LINE 2
<(<j,0>,<i,1>),1,11,main,9><(<n,0>),2,7,factorial,6><(),-,-,-,2>
FUNCTION factorial 2 7
< (<j,0>,<i,1>),1,11,main,9><(<n,0>),2,7,factorial,6><(),2,7,factorial,2>
FORMAL n 0
< (<j,0>,<i,1>),1,11,main,9><(<n,0>),2,7,factorial,6><(< n,0>),2,7,factorial,2>
LINE 3
< (<j,0>,<i,1>),1,11,main,9><(<n,0>),2,7,factorial,6><(< n,0>),2,7,factorial,3>
LOAD 0 n
LIT 2
BOP <
FALSEBRANCH else<<4>>
…

Notes:

- Each entry in the environment stack corresponds to a frame on the runtime stack
- Use the constrainer's symbol table mechanism to implement the symbol table in the environment stack entry.
- When a new environment stack entry is created you should create a new symbol table to include therein (create the symbol table and execute a *beginScope)*.
- The debugger has no indication when a new block is entered. Therefore, we cannot execute *beginScope's* in situations other than those mentioned in the prior point.
- The code generator generates a POP bytecode to pop all of the variables in the current scope (block). Therefore, we must pop all of the variables in our current scope (if we execute a POP 5 then we know that our current scope has 5 variables). This suggests we should remove the LAST 5 items entered in the symbol table – use code similar to that found in the *endScope* method to remove those symbols.

Example:

```
8. int f() {int j int k
9.       {int j    j = 1
10.     }
11.     j = 2
12. }
```

LABEL f<<2>>	<(),-,-,-,->	*create symbol table; beginScope*
LINE 8	<(),-,-,-,8>	
FUNCTION f 8 12	<(),8,12,f,8>	
LIT 0 j	<(<j,0>),8,12,f,8>	*enter j*
LIT 0 k	<(<j,0><k,1>),8,12,f,8>	*enter k*
LINE 9	<(<j,0><k,1>),8,12,f,9>	
LIT 0 j	<(<j,2><k,1>),8,12,f,9>	*enter j; symbol table will link to prior entry for j*
LIT 1		
STORE 2		
POP 1	<(<j,0><k,1>),8,12,f,9>	*remove last symbol entered, thereby restoring linked entry of j*
LINE 11	<(<j,0><k,1>),8,12,f,11>	
LIT 2		
STORE 0		
POP 2	<(),8,12,f,11>	*remove last 2 symbols*
RETURN f<<2>>		

The following are <u>hints</u> on implementing several of the debugger commands. Your implementations might vary somewhat from these implementation hints.

StepOut -- step out of the current function
boolean isSet -- if true then break just after function returns
int envSize -- this is the current function's Environment Stack size
--when the size decreases then we know we have stepped out

<when the current entry is popped then break>

StepInto -- step into the function on the current line
boolean isSet

<break if
- ***we hit a new line or***
- ***we push a new entry on the Environment Stack>***

StepOver -- step over the next line (DO NOT step into any functions in the line)
boolean isSet
int envSize -- this is the current function's Environment Stack size
--when the size decreases then we know we have stepped out

<break if
- ***the current line changes with Environment Stack size remaining the same as when the user requested the step out***
- ***when popping the current entry>***

<whenever we hit a breakpoint we'll *clear all* stepping flags>

Boolean isTraceOn -- if true then print trace information indented appropriately on function entry/exit

Sample trace output:
factorial(3) ← note that the args are in parens just like the function call
factorial(2) ← note the indentation
factorial(1)
exit factorial:1 ← note the indentation of the return matches the call indentation
exit factorial: 2
exit factorial: 6

Watching variables -- add a new *watch* option for the symbol table (change *Binder*); whenever we store into a variable we must check if the variable is being watched; if so, then print the variable name and it's new value

The Open/Closed Principle *(Bertrand Meyer)*

"Modules should be open for extension but closed for modification"

This principle has several ramifications with respect to your coding process. You can use this as a hint for coding the debugger. Rather than modifying much of the code you have already produced, you should consider using extension (what is extension??).
Hint: You do NOT want to place checks in much of your code to determine *if it's in debug mode then ... else ...* You can code in a *very clean fashion* without the *if* statements appearing in many places.

Additional Requirements:

1. Just after the Interpreter starts (before the VM executes the first instruction) you should allow the user to access all of the debugger commands to set breakpoints. Be sure to provide a help menu in a reasonable format.

You should also display the commands when the user types the help command.

Do not display the command list at any other time - e.g., when stopping at a breakpoint, **unless the user requests the command list by typing the *help* command.**

2. When you step into an intrinsic function (read/write) you should not display the source code. Just provide a note indicating which intrinsic function control is stopped in. You should stop, as appropriate, whether you're in an intrinsic function or not. Check what Netbeans does with library methods, you can step into those methods. In the debugger you don't have source code for the methods, so it is not possible to display source code. Whether you are in Netbeans or our debugger, if you step into a library/intrinsic function you probably want to simply step out to continue.

3. When you step into a read/write then you produce the usual information; however, since these are intrinsic functions the best you can do is print something like the following, without source code:

******READ*********

Whenever the debugger is given control, either during stepping or at a breakpoint, the source code should be displayed. Otherwise the user might have the wrong idea exactly where execution has stopped.

4. The debugger uses BOTH .x and .cod files. However, when you start the debugger you will provide the -d switch and only the BASE file name (e.g. factorial). The debugger will add the extensions when it starts running to get

factorial.x and factorial.x.cod

You should be able to run the debugger from a jar file:

java -jar interpreter.jar -d factorial

DO NOT ALTER THE EXPECTED COMMAND LINE ARGUMENTS.

Note, the program should run in "interpreter mode" if the user executes:

java -jar interpreter.jar factorial.x.cod

That is, the program might be run in either of 2 modes - normal interpreter and debugger if the -d switch is provided. You may assume if the program is run in normal interpreter mode then the bytecode file will not contain any debugger bytecodes (e.g., LINE).

5. Besides extending the *VirtualMachine*, think about what each bytecode does. Essentially, the bytecodes in the debugger do everything they would do in the interpreter, plus more. Not all bytecodes will need to be extended, however, you will have to extend bytecodes that affect the debugging environment.

Milestones

Exhaustive testing MUST be done at each milestone:
In the end I would like to execute a jar version of your program as:
java -jar interpreter.jar -d <base source file name> e.g.,
java -jar interpreter.jar -d factorial

This means that your program must check for the -d switch and provide the extensions to get factorial.x and factorial.x.cod. Also, these files should be found in the current directory. You should check this requirement prior to submitting a jar file.

Now, you can open a command window and execute the jar file as:
java -jar interpreter.jar -d factorial

Note, the program should run in "interpreter mode" if the user executes:

java -jar interpreter.jar factorial.x.cod

That is, ***the program might be run in either of 2 modes*** - normal interpreter and debugger if the -d switch is provided

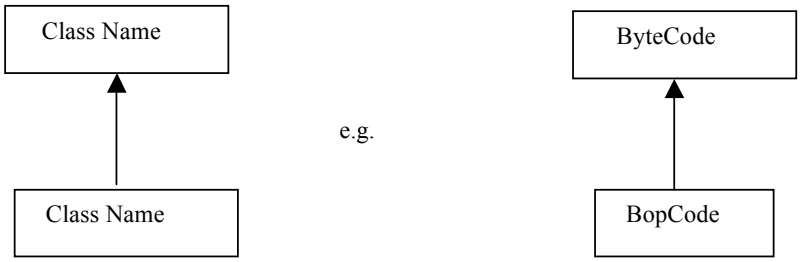
In addition to deliverables mentioned elsewhere, you will be required to **submit jar files for each major milestone (refer to Appendix A for notes on packaging your submissions)**.

Packages
- interpreter
  <previous interpreter project files>
- interpreter.bytecodes
  <bytecodes for original interpreter
- interpreter.bytecodes.debuggerByteCodes
  <new debugger bytecodes>
- interpreter.debugger
  <main debugger classes>
- interpreter.debugger.ui
  <classes dealing with the user interaction>

Milestone 1 (5 points): Submit Debugger design for approval
You will have your design evaluated either by a TA or Dr. Levine by attending an office hour and ask for your design to be graded. Alternatively, you can email your design to the assigned grader to evaluate your design – if you use this option then you will not get immediate feedback. If you submit your design by email then you should

Be sure to include your name and Milestone1 in the subject of the email. If all is well then it will be indicated. If there are problems then you will be notified and ***you will need to see me*** or the grader to discuss the problems and fix them. The design should consist of a class diagram depicting the main design issues – normally, each class picture should include class name, public or protected variables and public or protected methods. You do not need to draw the interpreter classes, if they are unchanged. Only draw new classes.
Exception to this rule: if you have a new debugger class A, which extends an interpreter class B, please draw A-->B (that is, please draw both A and B, even if B is unchanged, in case it has a new child A. Don't just draw A in those situations, because it makes your diagram unclear)

Use the following as a guide for constructing the class diagram (arrows go from a subclass to a superclass – DO NOT USE ANY OTHER ARROWS IN YOUR DIAGRAMS; ONLY PROVIDE THE CLASS NAMES, NOT METHODS OR VARIABLES FOR THIS LAB – there is a more complete example at the end of the Reader):



Milestone 2: FunctionEnvironmentRecord (8 points)

Implement/unit test basic infrastructure class:
FunctionEnvironmentRecord
Develop unit test for *FunctionEnvironmentRecord*; the *FunctionEnvironmentRecord* consists of *the symbol table, function name, starting/ending line number of the function and current line*. Provide getters/setters for the function name and lines. Then, test the symbol table mechanism, with revisions for the debugger as mentioned in class. In particular, this symbol table **should not be coupled to any other module**, it should only be capable of handling a single *beginscope*, *entering symbols/values* (Lit 0 i bytecodes) and *removing a given number of symbol/value bindings* (Pop bytecode). Note that the debugger uses the symbol table to record symbol/offset pairs where offset is the offset for the symbol on the runtime stack. However, for testing purposes do not make any assumptions regarding the values in symbol/value pairs – the values can be ANY *int* values (the *FunctionEnvironmentRecord* does not need to know how the values will be used).
Test the *FunctionEnvironmentRecord* by doing the following:

1. Include a main method in the *FunctionEnvironmentRecord* class (similar to unit tests I provided in the compiler modules) with the test case (see below).
2. Create an instance of *FunctionEnvironmentRecord (fctEnvRecord)*
3. **Process a series of commands** to test the unit; *after each symbol table command is issued dump the contents* of the *fctEnvRecord.* Note, since the symbol table uses a hashmap for its implementation, the order of symbol/value pairs you print out might not match the order in the following example.

Following are commands used for testing purposes:

BS	Begin scope
Function name start end	Set the fields for function name, start and end
Line n	Set the current line to n
Enter var value	Enter the var/value pair in the symbol table
Pop n	Remove the last n var/value pairs from the symbol table

Note, the API of FunctionEnvironmentRecord should include methods to perform the following operations: beginScope, setFunctionInfo, setCurrentLineNumber, setVarVal, and doPop.

TEST CASE
Following is the test case you should use – this test should be in the main method of *FunctionEnvironmentRecord.*

Command Issued	Dump	Comments
BS	(< >, -, -, -,-)	
Function g 1 20	(< >,g, 1, 20,-)	
Line 5	(< >,g, 1, 20,5)	
Enter a 4	(<a/4>, g, 1, 20, 5))	
Enter b 2	(<a/4,b/2>, g, 1, 20, 5)	
Enter c 7	(<a/4,c/7,b/2>, g, 1, 20, 5)	
Enter a 1	(<a/1, c/7,b/2>, g, 1, 20, 5)	The a/4 binding is not lost
Pop 2	(<a/4,b/2>, g, 1, 20, 5)	Delete the last 2 sym/value pairs entered
Pop 1	(<a/4>, g, 1, 20, 5)	

Submission

zip file as noted in Appendix A – zip the .jar file and FunctionEnvironmentRecord.java

Major Milestone 3 (15 points)

Implement the debugger so it executes the following commands
help, set/clear breakpoints, display the current function,
continue execution, quit execution
display variable(s)

Breakpoints may only be set at the beginning of the following constructs:
blocks decl if while return assign

e.g. for the following source program:

```
1.  program {boolean j int i
2.    int factorial(int n) {
3.       if (n < 2) then       <== The if node will have the boundaries of 3 and 6 to indicate that
4.          { return 1 }       <== the if construct starts at line 3 and ends at line 6.
5.       else                <== a breakpoint cannot be set on this line
6.          {return n*factorial(n-1) }
7.    }                      <== a breakpoint cannot be set on this line
8.    while (1==1) {
9.       i = write(factorial(read()))
10.   }
11.  }
```

Just after the Interpreter starts (before the VM executes the first instruction) you should print the source program and then prompt the user for a command; allow the user to access all of the debugger commands to set breakpoints. Be sure to provide a help menu in a reasonable format.

I suggest you prompt the user (in the UI) by providing the following:

Type ? for help
>

You should display the commands when the user types the *help* command.

Do not display the command list at any other time - e.g., when stopping at a breakpoint, **unless the user requests the command list by typing the *help* command.**

Submission

zip file as noted in Appendix A – zip the .jar file, src directory, fib.x and fib.x.cod

Milestone 4 (10 points): Submit zip file as noted in Appendix A

Implement and test step out – be careful to ensure step out returns from the ***current activation*** of the function.

Submission

zip file as noted in Appendix A – zip the .jar file, src, factorial.x, and factorial.x.cod

Major Milestone 5 (50 points); Submit zip file as noted in Appendix A:

Complete the entire lab, per ilearn specs for this milestone.

Note that if the user asks to display the source code of
the current function with execution stopped in the main program
then simply display the entire program.

Submission

You should turn in the following items as part of the zip file submission (Appendix A).

1. jar file
2. java source file directory structure (src)

Be sure to use Javadoc comments within your code (as is done in the code found herein); you will not turn in the result of running Javadoc since the source programs will be graded. If your interpreter project was not working correctly when it was

submitted then it is necessary for you to make corrections... NOTE: YOU MUST SUBMIT THIS LAB ON TIME – LATE LABS WILL NOT BE ACCEPTED!!

If your interpreter project was not working correctly when it was submitted then it is necessary for you to make corrections in order to continue with the debugger lab
NOTE: YOU MUST SUBMIT MILESTONE 5 ON TIME – LATE LABS WILL NOT BE ACCEPTED!!

The following gives an idea of an interaction with someone
using the debugger. I will not provide an exact dialogue since
you are required to think about an appropriate, professional
looking interaction.
Suppose, for example, that we use a double '>' as the debugger
prompt. I will use English for the commands - note that you
will provide appropriate command abbreviations (e.g. you might
use stpovr for step over). My comments are enclosed in <...>.
Most debugger commands have responses following the command (see
'set breakpoint' commands below).
Debugger output is enclosed in <<..>>, etc. Your debugger output should
not be enclosed in <<..>>
NOTE THAT WHENEVER EXECUTION STOPS DUE TO HITTING
A BREAKPOINT OR AS A RESULT OF A STEP INSTRUCTION, THE
CURRENT FUNCTION SHOULD BE PRINTED WITH STARS AND
ARROW, AS DEMONSTRATED BELOW.

listing with line numbers>
****Debugging factorial.x****
1. program {boolean j int i
2. int factorial(int n) {
3. if (n < 2) then
4. { return 1 }
5. else
6. {return n*factorial(n-1) }
7. }
8. while (1==1) {
9. i = write(factorial(read()))
10. }
11. }

Type ? for help
>>?

*present an appropriate help screen to the user; list the
debugger commands with VERY BRIEF explanations, if appropriate*

Type ? for help
>>set breakpoint at line 6

Breakpoint set: 6

Type ? for help
>>set breakpoint at line 9
Breakpoint set: 9

Type ? for help
>>continue execution

e.g. the program breaks at line 9,and gives the prompt
The source code for the current function (main program) is printed with line numbers - use good indentation; an indicator is included to mark the current line, along with indicator(s) for breakpoints, e.g.

```
1. program {boolean j int i
2. int factorial(int n) {
3. if (n < 2) then
4. { return 1 }
5. else
*6. {return n*factorial(n-1) }
7. }
8. while (1==1) {
*9. i = write(factorial(read())) <-----
10. }
11. }
```

the arrow indicates the current line being executed>

Type ? for help
>>clear breakpoint 6
Breakpoint cleared: 6

Type ? for help
>>set trace on

functions at function entry/exit as follows (you are not required to trace a specific function).

Type ? for help
>>continue execution
read()
Input an integer: 3
exit read: 3
factorial(3) note that the args are in parens just like the function call – there are no brackets

factorial(2) note the indentation; the number of spaces of indentation equals the size of the call
stack (equivalently, the size of the frame pointer stack or the size of the environment stack)
factorial(1)
exit factorial:1 note the indentation of the return matches the call indentation
exit factorial: 2
exit factorial: 6

6

e.g. breakpoint the program with line 9,and give the prompt
The source code for the current function (main program) is printed with line numbers - use good indentation; an indicator is included to mark the current line, along with indicator(s) for breakpoints, e.g.

```
1. program {boolean j int i
2. int factorial(int n) {
3. if (n < 2) then
4. { return 1 }
5. else
6. {return n*factorial(n-1) }
7. }
8. while (1==1) {
*9. i = write(factorial(read())) <-----
10. }
11. }
```

Type ? for help
>>set breakpoint at line 6
Breakpoint set: 6

Type ? for help
>> list breakpoints
Breakpoints: 6 9

Type ? for help
>>set trace off

Type ? for help
>>clear breakpoint 6
Breakpoint cleared: 6

Type ? for help
>>set breakpoint 3
Breakpoint set: 3

Type ? for help
>>continue execution

Input an integer: 5

2. int factorial(int n) {
**3. if (n < 2) then <------*
4. { return 1 }
5. else
*6. {return n*factorial(n-1) }*
7. }

Type ? for help
>>display local variables
n: 5

Type ? for help
>>list the source code for the current function
2. int factorial(int n) {
**3. if (n < 2) then <------*
4. { return 1 }
5. else
*6. {return n*factorial(n-1) }*
7. }

Type ? for help
>>step over
2. int factorial(int n) {
**3. if (n < 2) then*
4. { return 1 }
5. else
*6. {return n*factorial(n-1) } <------*
7. }

Type ? for help
>>step into
2. int factorial(int n) { <------
**3. if (n < 2) then*
4. { return 1 }
5. else
*6. {return n*factorial(n-1) }*
7. }

Type ? for help
>>display local variables

n: 4

>>clear breakpoint 3
Breakpoint cleared: 3

>>step out
2. int factorial(int n) {
3. if (n < 2) then
4. { return 1 }
5. else
*6. {return n*factorial(n-1) } <------*
7. }

Type ? for help
>>display local variables
n: 5

Type ? for help
>>Halt
*******Execution Halted*******

Following is a similar sample session without comments:

*****Debugging factorial.x*****

1. program {boolean j int i
2. int factorial(int n) {
3. if (n < 2) then
4. { return 1 }
5. else
*6. {return n*factorial(n-1) }*
7. }
8. while (1==1) {
9. i = write(factorial(read()))
10. }
11. }

Type ? for help
>>set breakpoint at lines 3 6 9
Breakpoints set: 3 6 9

Type ? for help
>>continue execution

1. program {boolean j int i
2. int factorial(int n) {
**3. if (n < 2) then*
4. { return 1 }
5. else
**6. {return n*factorial(n-1) }*
7. }
8. while (1==1) {
**9. i = write(factorial(read())) <--------*
10. }
11. }

Type ? for help
>>continue execution

Input an integer: 5

2. int factorial(int n) {
**3. if (n < 2) then <--------*
4. { return 1 }
5. else
**6. {return n*factorial(n-1) }*
7. }

Type ? for help
>> list breakpoints
Breakpoints: 3 6 9

Type ? for help
>>Display local variables
n: 5

Type ? for help
>>list source
2. int factorial(int n) {
**3. if (n < 2) then <--------*
4. { return 1 }
5. else
**6. {return n*factorial(n-1) }*
7. }

Type ? for help
>>clear breakpoint at line 3
Breakpoint cleared: 3

Type ? for help
>>continue execution
2. int factorial(int n) {
3. if (n < 2) then
4. { return 1 }
5. else
**6. {return n*factorial(n-1) } <--------*
7. }

Type ? for help
>>continue execution
2. int factorial(int n) {
3. if (n < 2) then
4. { return 1 }
5. else
**6. {return n*factorial(n-1) } <--------*
7. }

Type ? for help
>>display local variables
n: 4

Type ? for help
>>step out
2. int factorial(int n) {
3. if (n < 2) then
4. { return 1 }
5. else
**6. {return n*factorial(n-1) } <--------*
7. }

Type ? for help
>>display local variables
n: 3

Type ? for help
>>clear breakpoints
Breakpoints cleared: 6 9

Type ? for help
>>set breakpoint at line 9
Breakpoint set: 9

Type ? for help
>>continue execution

120

```
1. program {boolean j int i
2. int factorial(int n) {
3. if (n < 2) then
4. { return 1 }
5. else
6. {return n*factorial(n-1) }
7. }
8. while (1==1) {
*9. i = write(factorial(read())) <--------
10. }
11. }
```

Type ? for help
>>set trace on

Type ? for help
>>continue execution

```
Read()
Input an integer: 3
exit: Read: 3
factorial(3)
  factorial(2)
    factorial(1)
     exit: factorial: 1
  exit: factorial: 2
exit: factorial: 6
Write(6)
exit: Write: 6
```

6

```
1. program {boolean j int i
2. int factorial(int n) {
3. if (n < 2) then
4. { return 1 }
5. else
6. {return n*factorial(n-1) }
7. }
8. while (1==1) {
*9. i = write(factorial(read())) <--------
10. }
11. }
```

Type ? for help
>>Halt
******Execution Halted*******

Notes:

1. Be sure to provide comments in your code, as appropriate. You should minimize comments by using appropriate symbol names and provide clear code
2. There should only be a single active fetch/execute cycle. DO NOT include a fetch/execute loop with each step instruction.
3. You will need to create a package hierarchy as found in the Course Reader
4. All of the user interaction code should be included in the package *interpreter\debugger\UI*
5. You should use the following schema for your implementation:

a. When the user starts the debugger, after any initializations, the UI main interaction object should print the source program followed by a prompt to wait for user commands.

b. When the user is done interacting with the UI (e.g., she/he types continue) then the UI object should call the execute method of the debugger VM to start executing the bytecodes.

c. When a breakpoint is hit then the *execute* method should return so control resumes in the UI object to begin another user interaction session; this continues until the user wants more bytecodes executed by typing, e.g., *continue/step*, at which point the *execute* method will be called to execute bytecodes

Note that the *DebugVM* does not know anything about the UI – it's ***loosely coupled*** to the UI so it would be simple to replace the console UI with a GUI or internet-based UI. ***The DebugVM DOES NOT CALL THE UI, IT EXECUTES BYTECODES UNTIL IT HITS A BREAKPOINT OR STEP TARGET, ETC.***

Step In Note:

When you step into an intrinsic function (read/write) you should not display the source code. Just provide a note indicating which intrinsic function control is stopped in. You should stop, as appropriate, whether you're in an intrinsic function or not. Check what Netbeans does with library methods, you can step into those methods. In the debugger you don't have source code for the methods, so it is not possible to display source code. Whether you are in Netbeans or our debugger, if you step into a library/intrinsic function you probably want to simply step out to continue.

Trace Note:

When you step into a read/write then you produce the usual information; however, since these are intrinsic functions the best you can do is print something like the following, without source code:

******READ*********

Whenever the debugger is given control, either during stepping or at a breakpoint, the source code should be displayed. Otherwise the user might have the wrong idea exactly where execution has stopped.

Final Notes:

The debugger uses BOTH .x and .cod files. However, when you start the debugger you will provide the -d switch and only the BASE file name (e.g. factorial). The debugger will add the extensions when it starts running to get

factorial.x and factorial.x.cod

Besides extending the *VirtualMachine*, think about what each bytecode does. Essentially, the bytecodes in the debugger do everything they would do in the interpreter, plus more. Not all bytecodes will need to be extended, however, you will have to extend bytecodes that affect the debugging environment.

Q. Does the environment stack need to be printed in the debugger?

No. The environment stack never needs to be printed in the debugger. Dumping is a function of the interpreter only and should be disabled for the Debugger project.

Q. What does function tracing do?

You can consider function tracing to encompass 2 steps:

1. Printing the function name along with the values for the args (and the names of the args) **at function entry** – this should happen for ALL FUNCTION CALLS, INCLUDING INTRINSICS

2. Printing the function name along with the value being returned **at function exit**

Both 1 and 2 should be printed with proper indentation. If the function call is at level *n* in the function call stack then print it indented with (2**n*) spaces.

If tracing is turned on after function entry then the debugger should only print the step 2 information for that particular invocation and then print both step 1 and 2 information thereafter.

If you are checking a factorial function and are evaluating factorial of 20 then you will recurse 20 times. This means that you will see 20 function entry points (step 1) before you see the first function exit point...that is why it's important to use indentation when printing trace information..

Q. What does the Halt bytecode do in the debugger?

You should implement the HALT bytecode by setting the VM status to "not running". This should stop the debugger and exit the program gracefully.

Q. How should user input be handled when debugging?

Whenever you are running a program invoking a read function it should perform the

read (whether you are debugging the program or not).

Q. How are we using the symbol table from the Constrainer?

The reason for modifying it would be a situation like this:

```
{int i
point A. . .
  {int i int j
     point B. . .
     }
point C . . .
```

In this case, both of the i's and j are in the same FUNCTION;
therefore they are all in the same table. BUT, the second declaration of i is in a new BLOCK; therefore it should shadow the previous declaration of i until the end of that block.
At point C, the old i will resume.
SO, if the user is at point B and requests "display local variables", the value of the SECOND i should be displayed;
but if the user is at point A the value of the FIRST i should be displayed.
And all of this takes place within the SAME symbol table.
Also, if execution is stopped at point C and the user requests the variable values then the first i should be printed...
the j will not be in the table at that point..

Note that the *DebugVM* does not know anything about the UI – it's ***loosely coupled*** to the UI so it would be simple to replace the console UI with a GUI or internet-based UI. ***The DebugVM DOES NOT CALL THE UI, IT EXECUTES BYTECODES UNTIL IT HITS A BREAKPOINT OR STEP TARGET, ETC. ALL I/O, EXCEPT FOR READ/WRITE BYTECODES, IS PROVIDED BY THE UI. THIS MEANS, E.G., IF THE USER ASKS TO SEE THE SOURCE CODE OF THE CURRENT FUNCTION THEN THE REQUEST IS MADE TO THE DEBUG VM FOR THE SOURCE CODE (SINCE THE VM STORES THE SOURCE CODE) AND THE VM RETURNS A STRING THAT THE UI WILL SUBSEQUENTLY PRINT. CAREFULLY CONSIDER WHAT SHOULD BE INCLUDED IN THE VM API.***

The following rubrics should be considered when you are designing the debugger - your grade will be partially based on how well you address these issues:

1. Classes do not demonstrate code bloat.
2. Classes represent a single abstraction.
3. How well modules are loosely coupled?

4. How well classes/subclasses represent types/subtypes (consider the principle of substitutability)?

Final submission (Milestone 5): You should turn in the following items as part of the zip file submission (Appendix A).

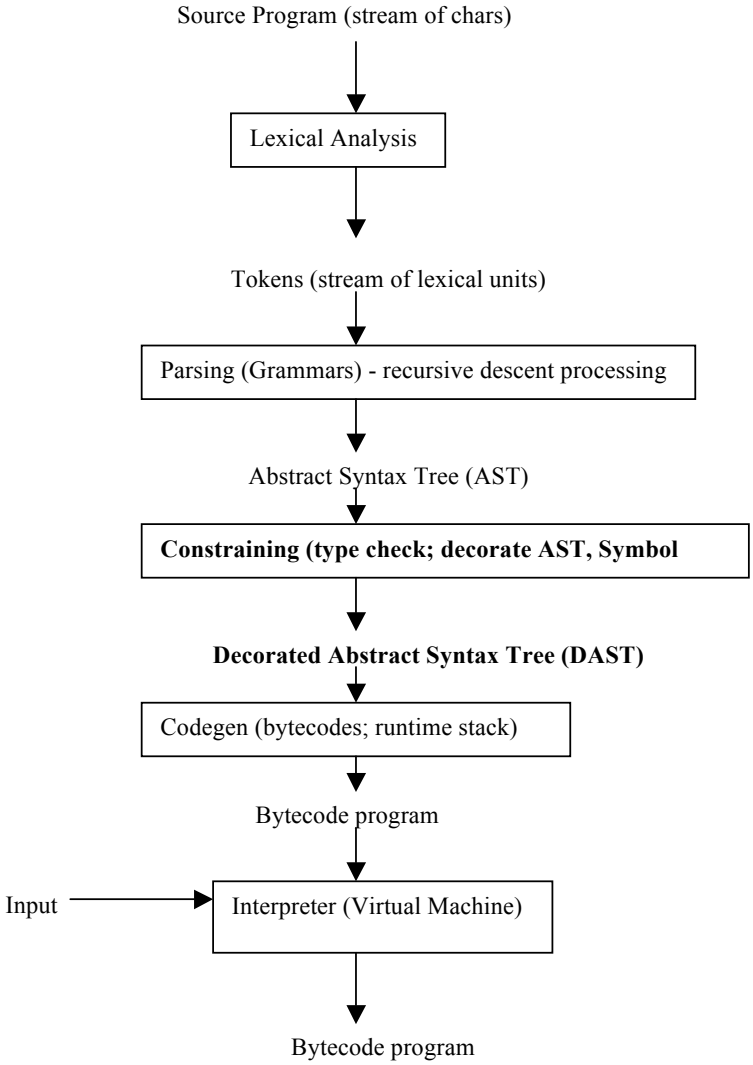
1. jar file
2. src directory from Netbeans

Be sure to use Javadoc comments within your code (as is done in the code found herein); you will not turn in the result of running Javadoc since the source programs will be graded.

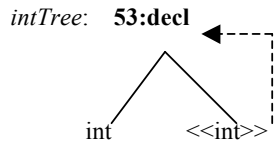
If your interpreter project was not working correctly when it was submitted then it is necessary for you to make corrections...

NOTE: YOU MUST SUBMIT MILESTONE 5 ON TIME – LATE LABS WILL NOT BE ACCEPTED!!

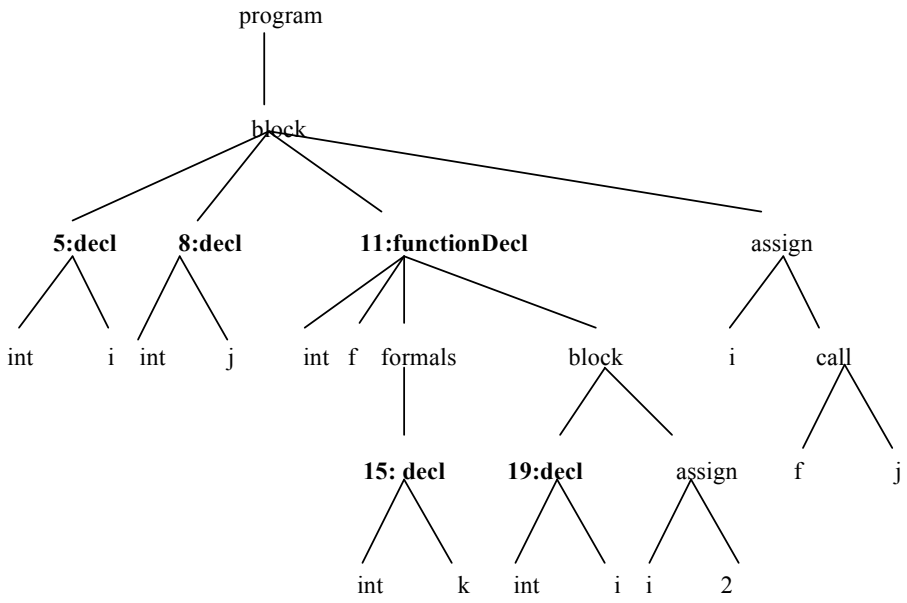
XIV. Constraining (Decorating the AST; Type Checking)



```
program { int i int j
  int f(int k) { int i
       i = 2
  }
 i = f(j)
}
```



AST:



<hand simulation using symbol table and constrainer activities>

scopes.x - Source program and Decorated AST (DAST)

```
program { int i int j
  int f(int i) { int j int k
       return i + j + k + 2
  }
  int m
  m = f(3)
  i = write(j + m)
}
```

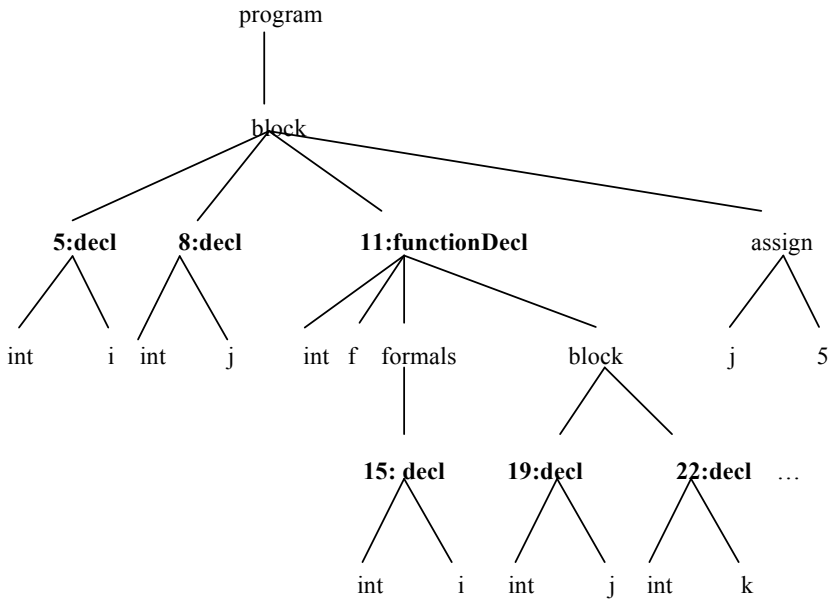
DECORATED AST for scopes.x

```
1:  Program
2:    Block
5:      Decl
3:        IntType
4:        Id: i          Dec: 53              53 is intTree
8:      Decl
6:        IntType
7:        Id: j          Dec: 53
11:     FunctionDecl
9:        IntType          Dec: 53
10:       Id: f
12:       Formals
15:         Decl
13:           IntType
14:           Id: i          Dec: 53
16:       Block
19:         Decl
17:           IntType
18:           Id: j          Dec: 53
22:         Decl
20:           IntType
21:           Id: k          Dec: 53
23:         Return           Dec: 11
29:           AddOp: +           Dec: 53
27:             AddOp: +           Dec: 53
25:               AddOp: +           Dec: 53
24:                 Id: i          Dec: 15
26:                 Id: j          Dec: 19
28:               Id: k          Dec: 22
30:             Int: 2           Dec: 53
33:     Decl
31:       IntType
32:       Id: m          Dec: 53
35:     Assign
34:       Id: m          Dec: 33
37:       Call           Dec: 53
36:         Id: f          Dec: 11
38:         Int: 3           Dec: 53
40:     Assign
39:       Id: i          Dec: 5
42:       Call           Dec: 53
41:         Id: write          Dec: 60          60 is FunctionDecl tree for write
```

44: AddOp: + Dec: 53
43: Id: j Dec: 8
45: Id: m Dec: 33

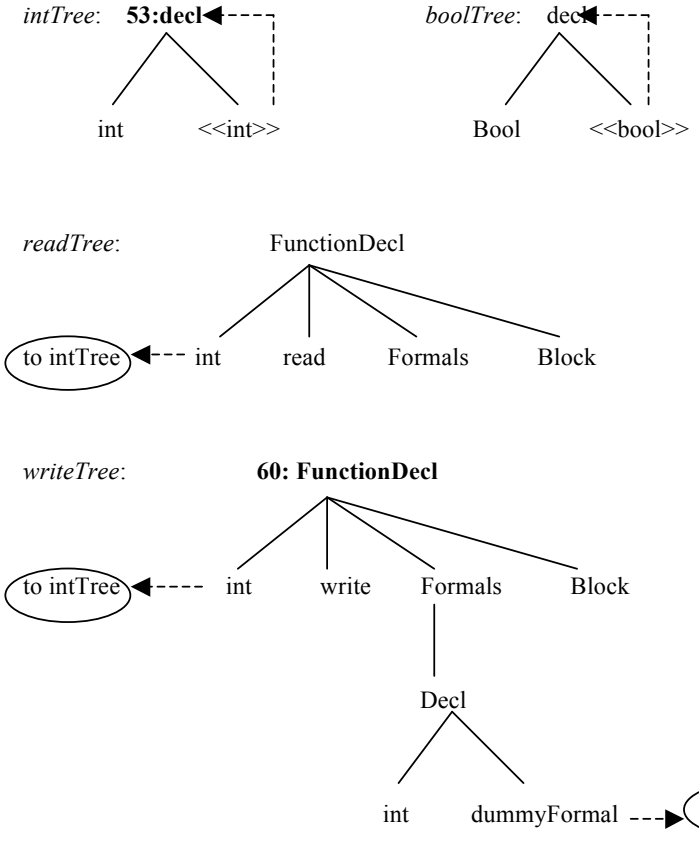
Note the following (partial) 2-dimensional tree

AST:



Note the need to record scopes for (i,j,f) then the formal i is in an inner scope (it *shadows* the outer i); then the block in the function has a new scope with (j,k) where the variable j shadows the outer j; after processing the functionDecl we need to remove the inner scopes to re-establish the scope (i,j,f) so when we use j in the assignment j = 5 we will refer to the outer j

Intrinsic Trees:



Variable Scopes and Symbol Tables

```
0.  execute                    beginScope
1:  Program
2:    Block                    beginScope
5:      Decl                   enter(i,5)   i is entered with link to Decl node
3:        IntType
4:        Id: i
8:      Decl                   enter(j,8)
6:        IntType
7:        Id: j
11:     FunctionDecl           enter(f,11), beginScope  Functions = 11
9:        IntType
10:       Id: f
12:       Formals
15:         Decl                 enter(i,15)
13:           IntType
14:           Id: i
16:       Block                beginScope
19:         Decl                 enter(j,19)
17:           IntType
18:           Id: j
22:         Decl                 enter(k,22)
20:           IntType
21:           Id: k
23:         Return               this return matches the function noted on top
                                 of the Functions stack (11 in this case)
```

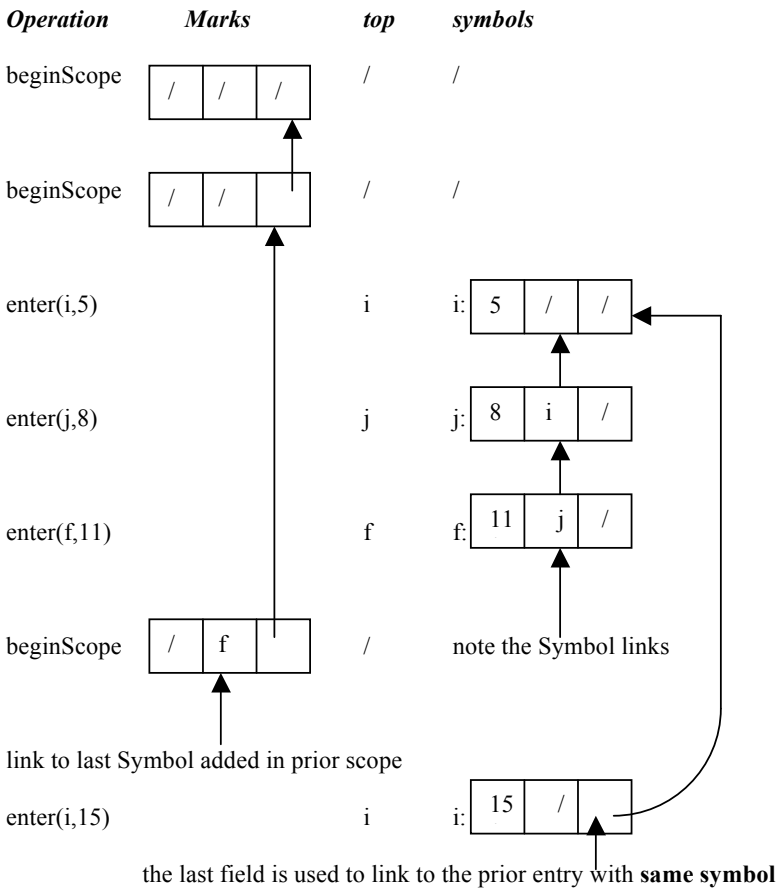
When we finish processing the Block node (node 16) we will call the endScope method, which will remove the symbol table entries in the current scope (j/19 and k/22) and re-establish the prior scope.

Scope Management

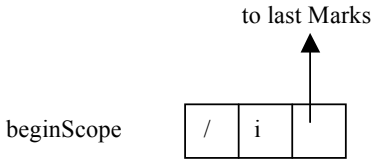
Marks: a stack of scope marks; each contains a link to the last variable entered in the symbol table at the current scope

top: the last Symbol entered in the Symbol table

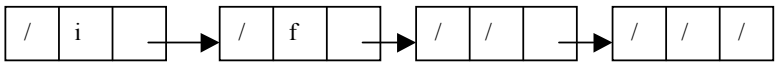
symbols: a HashMap where keys are Symbols and values are aggregations containing links for symbols in the current scope, links for entries with the same key and values for Symbols



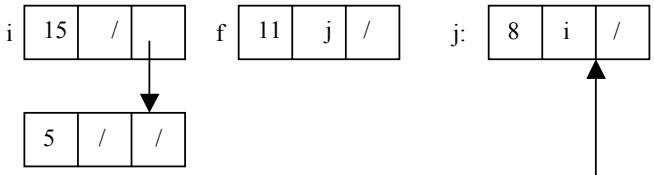
Operation ***Marks*** ***top*** ***symbols***



At this point we have:
Marks:



symbols:



Operation ***Marks*** ***top*** ***symbols***

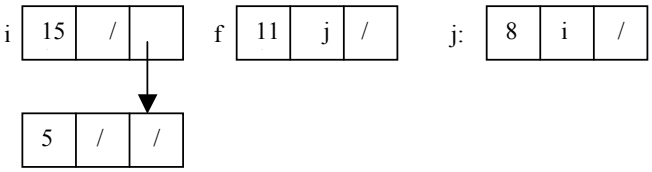
endScope (reset ***Marks*** to prior ***Marks***; reset ***top*** to last value in prior scope; remove all ***symbols*** entries in current scope; reset ***symbols*** entries to prior *Symbol* associations).

Since ***top*** has *j* we know that *j* was the last *Symbol* entered in current scope; we get its value in ***symbols*** and see that its *Symbol* link is null so there are no other entries in the current scope. We look at ***Marks*** to find that *i* was the top *Symbol* in the prior scope - we reset ***top*** to *i*.

If we look up *j* in ***symbols*** then we'll get the aggregate with the value 19
If we look up *i* in ***symbols*** then we'll get the aggregate with the value 15

After the endScope we have:
Marks:

symbols:



Note that the *prior j* entry in ***symbols*** is reset

symbols entries are:

keys are Symbols

values are *aggregates* (**Binder** instances - we are ***Bind***ing *Symbols* to these *aggregates*)

The aggregates include:

values - any Object

prior Symbol entered in *current* scope

prior Binder instance in some *prior* scope for same Symbol

Table.java

```
package constrain;

import lexer.Symbol;

/** <pre>
 *  Binder objects group 3 fields
 *  1. a value
 *  2. the next link in the chain of symbols in the current scope
 *  3. the next link of a previous Binder for the same identifier
 *     in a previous scope
 *  </pre>
*/
class Binder {
  private Object value;
  private Symbol prevtop;   // prior symbol in same scope
  private Binder tail;      // prior binder for same symbol
                            // restore this when closing scope
  Binder(Object v, Symbol p, Binder t) {
        value=v; prevtop=p; tail=t;
  }

  Object getValue() { return value; }
  Symbol getPrevtop() { return prevtop; }
  Binder getTail() { return tail; }
}


/** <pre>
 * The Table class is similar to java.util.Dictionary, except that
 * each key must be a Symbol and there is a scope mechanism.
 *
 * Consider the following sequence of events for table t:
 * t.put(Symbol("a"),5)
 * t.beginScope()
 * t.put(Symbol("b"),7)
 * t.put(Symbol("a"),9)
 *
 * symbols will have the key/value pairs for Symbols "a" and "b" as:
 *
 * Symbol("a") ->
 *     Binder(9, Symbol("b") , Binder(5, null, null) )
 * (the second field has a reference to the prior Symbol added in this
 * scope; the third field refers to the Binder for the Symbol("a")
 * included in the prior scope)
 * Binder has 2 linked lists - the second field contains list of symbols
 * added to the current scope; the third field contains the list of
 * Binders for the Symbols with the same string id - in this case, "a"
 *
 * Symbol("b") ->
 *     Binder(7, null, null)
 * (the second field is null since there are no other symbols to link
 * in this scope; the third field is null since there is no Symbol("b")
```

```
 * in prior scopes)
 *
 * top has a reference to Symbol("a") which was the last symbol added
 * to current scope
 *
 * Note: What happens if a symbol is defined twice in the same scope??
 * </pre>
*/
public class Table {

  private java.util.HashMap<Symbol,Binder> symbols = new
java.util.HashMap<Symbol,Binder>();
  private Symbol top;    // reference to last symbol added to
                         // current scope; this essentially is the
                         // start of a linked list of symbols in scope
  private Binder marks;  // scope mark; essentially we have a stack of
                         // marks - push for new scope; pop when closing
                         // scope

/*
   public static void main(String args[]) {
      Symbol s = Symbol.symbol("a", 1),
         s1 = Symbol.symbol("b", 2),
         s2 = Symbol.symbol("c", 3);

      Table t = new Table();
      t.beginScope();
      t.put(s,"top-level a");
      t.put(s1,"top-level b");
      t.beginScope();
      t.put(s2,"second-level c");
      t.put(s,"second-level a");
      t.endScope();
      t.put(s2,"top-level c");
      t.endScope();
}

*/
  public Table(){}


 /**
  * Gets the object associated with the specified symbol in the Table.
  */
  public Object get(Symbol key) {
       Binder e = symbols.get(key);
       return e.getValue();
  }

 /**
  * Puts the specified value into the Table, bound to the specified Symbol.<br>
  * Maintain the list of symbols in the current scope (top);<br>
  * Add to list of symbols in prior scope with the same string identifier
  */
  public void put(Symbol key, Object value) {
       symbols.put(key, new Binder(value, top, symbols.get(key)));
```

```
        top = key;
    }

    /**
     * Remembers the current state of the Table; push new mark on mark stack
     */
    public void beginScope() {
      marks = new Binder(null,top,marks);
      top=null;
    }

    /**
     * Restores the table to what it was at the most recent beginScope
     *     that has not already been ended.
     */
    public void endScope() {
        while (top!=null) {
          Binder e = symbols.get(top);
          if (e.getTail()!=null) symbols.put(top,e.getTail());
          else symbols.remove(top);
          top = e.getPrevtop();
        }
        top=marks.getPrevtop();
        marks=marks.getTail();
    }

    /**
     * @return a set of the Table's symbols.
     */
    public java.util.Set<Symbol> keys() {return symbols.keySet();}
}
```

```
program { int i int j
  int f(int i) { int j int k
       return i + j + k + 2
  }
  int m
  m = f(3)
  i = write(j + m)
}
```

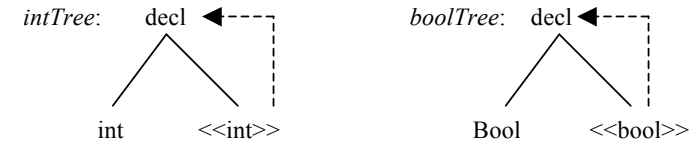
DECORATED AST for scopes.x

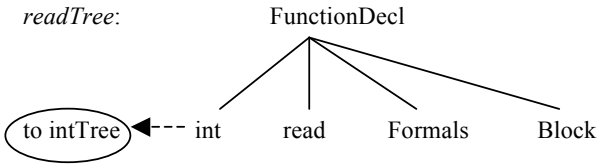
```
1:  Program
2:    Block
5:      Decl
3:        IntType
4:        Id: i          Dec: 53              53 is intTree
8:      Decl
6:        IntType
7:        Id: j          Dec: 53
11:     FunctionDecl
9:        IntType          Dec: 53
10:       Id: f
12:       Formals
15:         Decl
13:           IntType
14:           Id: i          Dec: 53
16:       Block
19:         Decl
17:           IntType
18:           Id: j          Dec: 53
22:         Decl
20:           IntType
21:           Id: k          Dec: 53
23:         Return          Dec: 11            11 is FunctionDecl for current function
29:           AddOp: +          Dec: 53
27:             AddOp: +          Dec: 53
25:               AddOp: +          Dec: 53
24:                 Id: i          Dec: 15        15 is Decl node for i
26:                 Id: j          Dec: 19
28:               Id: k          Dec: 22
30:             Int: 2          Dec: 53
33:     Decl
31:       IntType
32:       Id: m          Dec: 53
35:     Assign
34:       Id: m          Dec: 33
37:       Call          Dec: 53
36:         Id: f          Dec: 11
38:         Int: 3          Dec: 53
40:     Assign
39:       Id: i          Dec: 5
42:       Call          Dec: 53
41:         Id: write          Dec: 60              60 is FunctionDecl tree for write
44:         AddOp: +          Dec: 53
43:           Id: j          Dec: 8
```

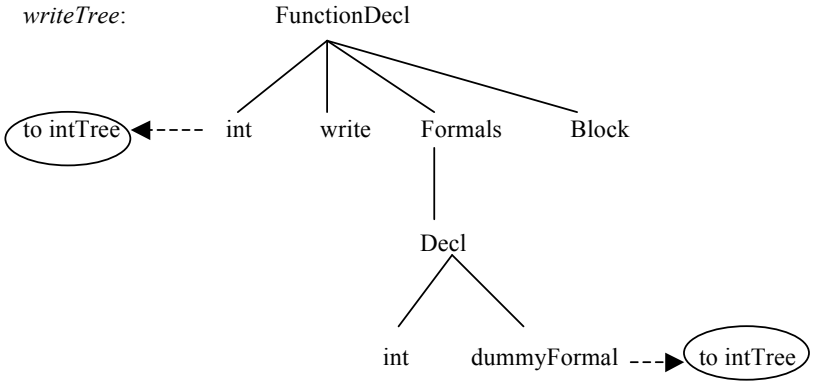
Constraining Activities:

- Type checking
- Decorates each type to the *intrinsic* type tree
- Decorates all variable references back to their Decl trees
- Decorate each return to its associated FunctionDecl tree

Intrinsic Trees:








```
/**
 *  build the intrinsic trees; constrain them in the same fashion
 *  as any other AST
*/
   private void buildIntrinsicTrees() {
      Lexer lex = parser.getLex();
      ...
      intTree = (new DeclTree()).addKid(new IntTypeTree()).
            addKid(new IdTree(lex.newIdToken("<<int>>",-1,-1)));
      decorate(intTree.getKid(2),intTree);
      writeTree = (new FunctionDeclTree()).addKid(new IntTypeTree()).
               addKid(writeId);
      AST decl = (new DeclTree()).addKid(new IntTypeTree()).
               addKid(new IdTree(lex.newIdToken("dummyFormal",-1,-1)));
      AST formals = (new FormalsTree()).addKid(decl);
      writeTree.addKid(formals).addKid(new BlockTree());
      writeTree.accept(this);   //constrain writeTree just as we would constrain any
                                //user defined function
   }
```

Processing by the Constrainer

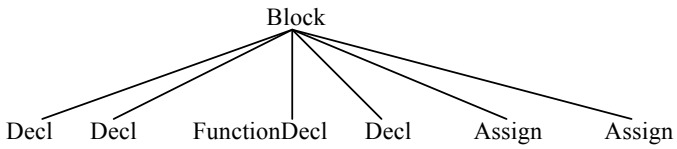
Simulate the constrainer with:

```
program {int i
  int f(int j) {int i return j+5}
  i = f(7)
}
```

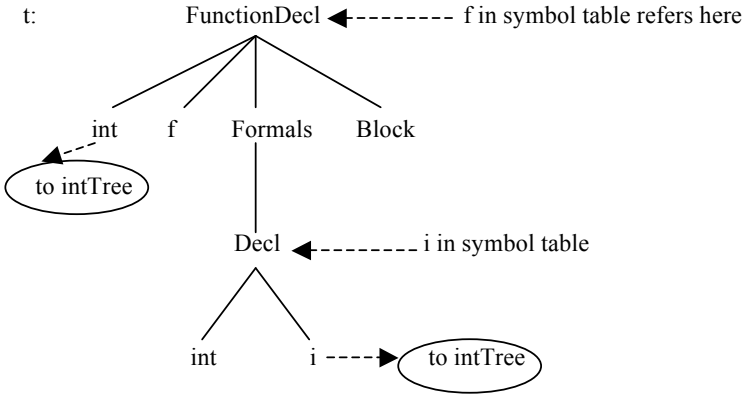
Program

Block

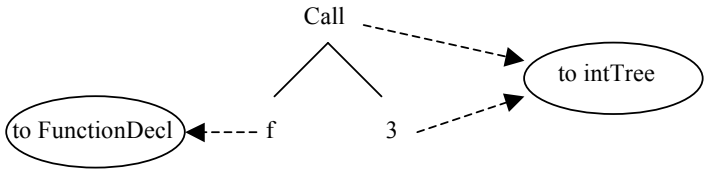
```
  public Object visitProgramTree(AST t) {
    buildIntrinsicTrees();
    this.t = t;
    t.getKid(1).accept(this);
    return null;
  }
```



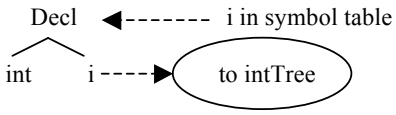

```
  public Object visitBlockTree(AST t) {
    symtab.beginScope();
    visitKids(t);
    symtab.endScope();
    return null; }
```



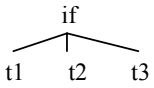

```
public Object visitFunctionDeclTree(AST t) {
      AST fname = t.getKid(2),
         returnType = t.getKid(1),
         formalsTree = t.getKid(3),
         bodyTree = t.getKid(4);
      functions.push(t);
      enter(fname,t);  // enter function name in CURRENT scope
      decorate(returnType,getType(returnType));
      symtab.beginScope();  // new scope for formals and body
      visitKids(formalsTree); // all formal names go in new scope
      bodyTree.accept(this);
      symtab.endScope();
      functions.pop();
```



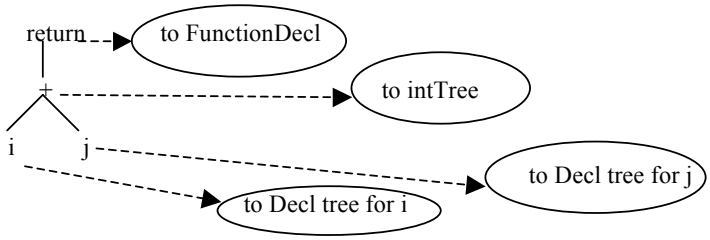

```
   public Object visitCallTree(AST t) {
      AST fct,
         fname = t.getKid(1),
         fctType;
      visitKids(t);
      fct = lookup(fname);
      if (fct.getClass() != FunctionDeclTree.class) {
        constraintError(ConstrainerErrors.CallingNonFunction);
      }
      fctType = decoration(fct.getKid(1));
      decorate(t,fctType);     decorate(t.getKid(1),fct);
      // now check that number/types of actuals match the number/types of formals
      checkArgDecls(t,fct);    return fctType;
```



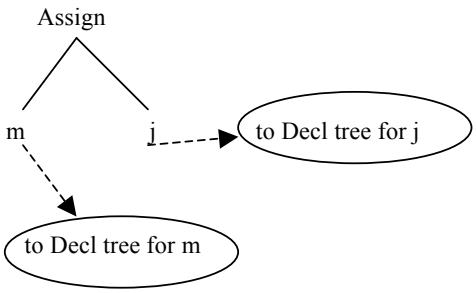

```
public Object visitDeclTree(AST t) {
   AST idTree = t.getKid(2);
   enter(idTree,t);
   AST typeTree = getType(t.getKid(1));    decorate(idTree,typeTree);
```




```
public Object visitIfTree(AST t) {
   AST kid;
   if ( (AST)t.getKid(1).accept(this) != boolTree) {
      constraintError(ConstrainerErrors.BadConditional);
   }
   t.getKid(2).accept(this);
   t.getKid(3).accept(this);
```




```
*  Check that the returned expression type matches the type indicated
*  in the function we're returning from
   public Object visitReturnTree(AST t) {
      if (functions.empty()) {
         constraintError(ConstrainerErrors.ReturnNotInFunction);
       }
      AST currentFunction = (AST)(functions.peek());
      decorate(t,currentFunction);
      AST returnType = decoration(currentFunction.getKid(1));
      if ( (AST)(t.getKid(1).accept(this)) != returnType) {
         constraintError(ConstrainerErrors.BadReturnExpr);
      }
```




```
public Object visitAssignTree(AST t) {
   AST idTree = t.getKid(1),
         idDecl = lookup(idTree),
         typeTree;
   decorate(idTree,idDecl);
   typeTree = decoration(idDecl.getKid(2));

   // now check that the types of the expr and id are the same
   // visit the expr tree and get back its type
   if ( (AST)(t.getKid(2).accept(this)) != typeTree) {
         constraintError(ConstrainerErrors.BadAssignmentType);
  }
   return null;
}
```

Constrainer.java

```
package constrain;

import lexer.*;
import parser.Parser;
import visitor.*;
import ast.*;
import java.util.*;

/**
 *  Constrainer object will visit the AST, gather/check variable
 *  type information and decorate uses of variables with their
 *  declarations; the decorations will be used by the code generator
 *  to provide access to the frame offset of the variable for generating
 *  load/store bytecodes; <br>
 *  Note that when constraining expression trees we return the type tree
 *  corresponding to the result type of the expression; e.g.
 *  the result of constraining the tree for 1+2*3 will be the int type
 *  tree
*/
public class Constrainer extends ASTVisitor {
        public enum ConstrainerErrors {
           BadAssignmentType, CallingNonFunction, ActualFormalTypeMismatch,
           NumberActualsFormalsDiffer, TypeMismatchInExpr,
           BooleanExprExpected, BadConditional, ReturnNotInFunction, BadReturnExpr
        }



    private AST t;          // the AST to constrain
    private Table symtab = new Table();
    private Parser parser;   // parser used with this constrainer

/**
 *  The following comment refers to the functions stack
 *  declared below the comment.
 *  Whenever we start constraining a function declaration
 *  we push the function decl tree which indicates we're
 *  in a function (to ensure that we don't attempt to return
 *  from the main program - return's are only allowed from
 *  within functions); it also gives us access to the return
 *  type to ensure the type of the expr that is returned is
 *  the same as the type declared in the function header
*/
    private Stack<AST> functions = new Stack<AST>();

/**
 *  readTree, writeTree, intTree, boolTree,falseTree, trueTree
 *  are AST's that will be constructed (intrinsic trees) for
 *  every program. They are constructed in the same fashion as
 *  source program trees to ensure consisten processing of
 *  functions, etc.
*/
```

```
   public static AST readTree, writeTree, intTree, boolTree,
          falseTree, trueTree, readId, writeId;

   public Constrainer(AST t, Parser parser) {
      this.t = t;
      this.parser = parser;
   }

   public void execute() {
      symtab.beginScope();
      t.accept(this);
   }

/**
 *  t is an IdTree; retrieve the pointer to its declaration
*/
   private AST lookup(AST t) {
      return (AST)(symtab.get( ((IdTree)t).getSymbol()));
   }

/**
 *  Decorate the IdTree with the given decoration - its decl tree
*/
   private void enter(AST t, AST decoration) {
/*
      System.out.println("enter: "+((IdTree)t).getSymbol().toString()
          + ": " + decoration.getNodeNum());
*/
      symtab.put( ((IdTree)t).getSymbol(), decoration);
   }

/**
 *  get the type of the current type tree
 *  @param t is the type tree
 *  @return the intrinsic tree corresponding to the type of t
*/
   private AST getType(AST t) {
      return (t.getClass() == IntTypeTree.class)? intTree: boolTree;
   }

   public void decorate(AST t, AST decoration) {
      t.setDecoration(decoration);
   }

/**
 *  @return the decoration of the tree
*/
   public AST decoration(AST t) {
      return t.getDecoration();
   }

/**
 *  build the intrinsic trees; constrain them in the same fashion
 *  as any other AST
*/
   private void buildIntrinsicTrees() {
```

```
        Lexer lex = parser.getLex();
        trueTree = new IdTree(lex.newIdToken("true",-1,-1));
        falseTree = new IdTree(lex.newIdToken("false",-1,-1));
        readId = new IdTree(lex.newIdToken("read",-1,-1));
        writeId = new IdTree(lex.newIdToken("write",-1,-1));
        boolTree = (new DeclTree()).addKid(new BoolTypeTree()).
                addKid(new IdTree(lex.newIdToken("<<bool>>",-1,-1)));
        decorate(boolTree.getKid(2),boolTree);
        intTree = (new DeclTree()).addKid(new IntTypeTree()).
                addKid(new IdTree(lex.newIdToken("<<int>>",-1,-1)));
        decorate(intTree.getKid(2),intTree);
        // to facilitate type checking; this ensures int decls and id decls
        // have the same structure

        // read tree takes no parms and returns an int
        readTree = (new FunctionDeclTree()).addKid(new IntTypeTree()).
                        addKid(readId).addKid(new FormalsTree()).
                        addKid(new BlockTree());

        // write tree takes one int parm and returns that value
        writeTree = (new FunctionDeclTree()).addKid(new IntTypeTree()).
                        addKid(writeId);
        AST decl = (new DeclTree()).addKid(new IntTypeTree()).
                        addKid(new IdTree(lex.newIdToken("dummyFormal",-1,-1)));
        AST formals = (new FormalsTree()).addKid(decl);
        writeTree.addKid(formals).addKid(new BlockTree());
        writeTree.accept(this);
        readTree.accept(this);
    }

/**
 *  Constrain the program tree - visit its kid
*/
    public Object visitProgramTree(AST t) {
        buildIntrinsicTrees();
        this.t = t;
        t.getKid(1).accept(this);
        return null;
    }

/**
 *  Constrain the Block tree:<br>
 *  <ol><li>open a new scope, <li>constrain the kids in this new scope, <li>close the
 *  scope removing any local declarations from this scope</ol>
*/
    public Object visitBlockTree(AST t) {
        symtab.beginScope();
        visitKids(t);
        symtab.endScope();
        return null; }

/**
 *  Constrain the FunctionDeclTree:
 *  <ol><li>Enter the function name in the current scope, <li>enter the formals
 *  in the function scope and <li>constrain the body of the function</ol>
*/
```

```
   public Object visitFunctionDeclTree(AST t) {
      AST fname = t.getKid(2),
          returnType = t.getKid(1),
          formalsTree = t.getKid(3),
          bodyTree = t.getKid(4);
      functions.push(t);
      enter(fname,t);  // enter function name in CURRENT scope
      decorate(returnType,getType(returnType));
      symtab.beginScope();  // new scope for formals and body
      visitKids(formalsTree); // all formal names go in new scope
      bodyTree.accept(this);
      symtab.endScope();
      functions.pop();
      return null;
   }

/**
 *  Constrain the Call tree:<br>
 *  check that the number and types of the actuals match the
 *  number and type of the formals
*/
   public Object visitCallTree(AST t) {
      AST fct,
          fname = t.getKid(1),
          fctType;
      visitKids(t);
      fct = lookup(fname);
      if (fct.getClass() != FunctionDeclTree.class) {
         constraintError(ConstrainerErrors.CallingNonFunction);
      }
      fctType = decoration(fct.getKid(1));
      decorate(t,fctType);
      decorate(t.getKid(1),fct);
      // now check that the number/types of actuals match the
      // number/types of formals
      checkArgDecls(t,fct);
      return fctType;
   }

   private void checkArgDecls(AST caller, AST fct) {
      // check number and types of args/formals match
      AST formals = fct.getKid(3);
      Iterator<AST> actualKids = caller.getKids().iterator(),
                    formalKids = formals.getKids().iterator();
      actualKids.next();  // skip past fct name
      for (; actualKids.hasNext();) {
         try {
            AST actualDecl = decoration(actualKids.next()),
                formalDecl = formalKids.next();
            if (decoration(actualDecl.getKid(2)) !=
                decoration(formalDecl.getKid(2))) {
                constraintError(ConstrainerErrors.ActualFormalTypeMismatch);
            }
         } catch (Exception e) {
            constraintError(ConstrainerErrors.NumberActualsFormalsDiffer);
         }
```

```
        }
        if (formalKids.hasNext()) {
            constraintError(ConstrainerErrors.NumberActualsFormalsDiffer);
        }
        return;
    }

/**
 *  Constrain the Decl tree:<br>
 *  <ol><li>decorate to the corresponding intrinsic type tree, <li>enter the
 *  variable in the current scope so later variable references can
 *  retrieve the information in this tree</ol>
*/
    public Object visitDeclTree(AST t) {
        AST idTree = t.getKid(2);
        enter(idTree,t);
        AST typeTree = getType(t.getKid(1));
        decorate(idTree,typeTree);
        return null; }

/**
 *  Constrain the <i>If</i> tree:<br>
 *  check that the first kid is an expression that is a boolean type
*/
    public Object visitIfTree(AST t) {
        if ( t.getKid(1).accept(this) != boolTree) {
            constraintError(ConstrainerErrors.BadConditional);
        }
        t.getKid(2).accept(this);
        t.getKid(3).accept(this);
        return null;
    }

    public Object visitWhileTree(AST t) {
        if ( t.getKid(1).accept(this) != boolTree) {
            constraintError(ConstrainerErrors.BadConditional);
        }
        t.getKid(2).accept(this);
        return null;
    }

/**
 *  Constrain the Return tree:<br>
 *  Check that the returned expression type matches the type indicated
 *  in the function we're returning from
*/
    public Object visitReturnTree(AST t) {
        if (functions.empty()) {
            constraintError(ConstrainerErrors.ReturnNotInFunction);
        }
        AST currentFunction = (functions.peek());
        decorate(t,currentFunction);
        AST returnType = decoration(currentFunction.getKid(1));
        if ( (t.getKid(1).accept(this)) != returnType) {
            constraintError(ConstrainerErrors.BadReturnExpr);
        }
```

```
      return null;
   }

/**
 *  Constrain the Assign tree:<br>
 *  be sure the types of the right-hand-side expression and variable
 *  match; when we constrain an expression we'll return a reference
 *  to the intrinsic type tree describing the type of the expression
*/
   public Object visitAssignTree(AST t) {
      AST idTree = t.getKid(1),
         idDecl = lookup(idTree),
         typeTree;
      decorate(idTree,idDecl);
      typeTree = decoration(idDecl.getKid(2));

      // now check that the types of the expr and id are the same
      // visit the expr tree and get back its type
      if ( (t.getKid(2).accept(this)) != typeTree) {
         constraintError(ConstrainerErrors.BadAssignmentType);
      }
      return null;
   }

   public Object visitIntTree(AST t) {
      decorate(t,intTree);
      return intTree;
   }

   public Object visitIdTree(AST t) {
      AST decl = lookup(t);
      decorate(t,decl);
      return decoration(decl.getKid(2));
   }

   public Object visitRelOpTree(AST t) {
      AST leftOp = t.getKid(1),
         rightOp = t.getKid(2);
      if ( (AST)(leftOp.accept(this)) != (AST)(rightOp.accept(this)) ) {
         constraintError(ConstrainerErrors.TypeMismatchInExpr);
      }
      decorate(t,boolTree);
      return boolTree;
   }

/**
 *  Constrain the expression tree with an adding op at the root:<br>
 *  e.g. t1 + t2<br>
 *  check that the types of t1 and t2 match, if it's a plus tree
 *  then the types must be a reference to the intTree
 *  @return the type of the tree
*/
   public Object visitAddOpTree(AST t) {
      AST leftOpType = (AST)(t.getKid(1).accept(this)),
         rightOpType = (AST)(t.getKid(2).accept(this));
      if (leftOpType != rightOpType) {
```

```
            constraintError(ConstrainerErrors.TypeMismatchInExpr);
        }
        decorate(t,leftOpType);
        return leftOpType;
    }

    public Object visitMultOpTree(AST t) {
        return visitAddOpTree(t);
    }

    public Object visitIntTypeTree(AST t) {return null;}
    public Object visitBoolTypeTree(AST t) {return null;}
    public Object visitFormalsTree(AST t) {return null;}
    public Object visitActualArgsTree(AST t) {return null;}

    void constraintError(ConstrainerErrors err) {
        PrintVisitor v1 = new PrintVisitor();
        v1.visitProgramTree(t);
        System.out.println("****CONSTRAINER ERROR: " + err + "    ****");
        System.exit(1);
        return;
    }

}
```

DECORATED AST for scopes.x

```
1:  Program
2:    Block
5:      Decl
3:        IntType
4:        Id: i          Dec: 53              53 is intTree
8:      Decl
6:        IntType
7:        Id: j          Dec: 53
11:     FunctionDecl
9:        IntType          Dec: 53
10:       Id: f
12:       Formals
15:         Decl
13:           IntType
14:           Id: i          Dec: 53
16:       Block
19:         Decl
17:           IntType
18:           Id: j          Dec: 53
22:         Decl
20:           IntType
21:           Id: k          Dec: 53
23:         Return          Dec: 11
29:           AddOp: +          Dec: 53
27:             AddOp: +          Dec: 53
25:               AddOp: +          Dec: 53
24:                 Id: i          Dec: 15
26:                 Id: j          Dec: 19
28:               Id: k          Dec: 22
30:             Int: 2          Dec: 53
33:     Decl
31:       IntType
32:       Id: m          Dec: 53
35:     Assign
34:       Id: m          Dec: 33
37:       Call          Dec: 53
36:         Id: f          Dec: 11
38:         Int: 3          Dec: 53
40:     Assign
39:       Id: i          Dec: 5
42:       Call          Dec: 53
41:         Id: write          Dec: 60          60 is FunctionDecl tree for write
44:         AddOp: +          Dec: 53
43:           Id: j          Dec: 8
45:           Id: m          Dec: 33
```

XV. Code Generation

Frames (Activation Records) and the Runtime stack

1. Walk the *DAST*
2. Generate *bytecodes*
3. Track how each code affects the runtime stack - whenever we determine a frame offset for a variable declaration, set the address field with the value

Bytecode classes:

Code.java

```
package codegen;

/** The Code class records bytecode information
*/
public class Code {
   private Codes.ByteCodes bytecode;

   public Code(Codes.ByteCodes code) {
      bytecode = code;
   }

   public Codes.ByteCodes getBytecode() {
      return bytecode;
   }

   public String toString() {
      return bytecode.toString();
   }

   public void print() {
      System.out.println(toString());
   }
}

/**
 * LabelOpcode class records bytecodes with associated labels
 * e.g. LABEL xyz
*/
class LabelOpcode extends Code {
   String label;

/**
```

```
 *  @param code is the bytecode being created
 *  @param label is the string representation of the label of interest
*/
   public LabelOpcode(int code, String label) {
      super(code);
      this.label = label;
   }

   public void print() {
      System.out.println(toString());
   }

   public String toString() {
      return super.toString() + " " + label;
   }
}

/** NumOpcode class used for bytecodes with a number op field
 *  e.g. lit 5
*/
class NumOpcode extends Code {
   int num;

   public NumOpcode(int code, int n) {
      super(code);
      num = n;
   }

   int getNum() {
      return num;
   }

   public String toString() {
      return super.toString() + " " + num;
   }

   public void print() {
      System.out.println(toString());
   }
}

/**
 *  VarOpcode class used for bytecodes with addresses
 *  and string information - e.g. LOAD 0 x
*/
class VarOpcode extends Code {
   int location;
   String varname;

/**
 *  @param code is the bytecode being created
 *  @param location is the offset from the start of the current frame
 *  @param varname is the name of the variable being loaded/stored
*/
   public VarOpcode(int code, int location, String varname) {
      super(code);
```

```
        this.location = location;
        this.varname = varname;
    }

    public String toString() {
        return super.toString() + " " + location + " " + varname;
    }

    public void print() {
        System.out.println(toString());
    }
}
```

Program.java

```
package codegen;

import java.util.*;
import java.io.*;

/**
 * This class will hold the generated program bytecodes
*/
public class Program {
   private ArrayList<Code> program = new ArrayList<Code>();

/**
 *  store the new bytecode in the program vector
 *  @param code is the bytecode to store
*/
   public void storeop(Code code) {
      program.add(code);
   }

/**
 *  print all of the bytecodes that have been generated
 *  @param outFile a String indicating where to print the bytecodes
*/
   public void printCodes(String outFile) {
      PrintWriter out = null;
      try {
          out = new PrintWriter(new FileOutputStream(outFile));
      } catch (IOException e) {
         System.out.println(e.toString());
         System.exit(1);
      }

      for (Code nextCode : program) {
           System.out.println(nextCode.toString());
         out.println(nextCode.toString());
      }
      out.close();
   }
}
```

Frames: associated with functions and the main program

Blocks: associated with program blocks

```
program  {  int i  int j     Variable        Offset in Frame  Offset in Block
                             i               0                0
                             j               1                1
     i = 5
     {  int  k               k               2                0
        k = 5   j = k
     }
     {  int m                m               2                0
        m = 7   i = m
     }                                <when a block ends its variables are removed
                                       from the runtime stack>
```

Tracking Frame and Block Sizes

We need to track the Frame and Block sizes in order to determine the variable offsets. Since we enter/exit Frames and Blocks in LIFO fashion we'll use stacks to record their sizes at all times during execution.

1. **Add new variable in a block** (e.g. int j)

   Bump both the block and frame sizes

2. **Enter a Block**

   Push a new block size tracker on the stack; init the size to 0

3. **Leave a Block**

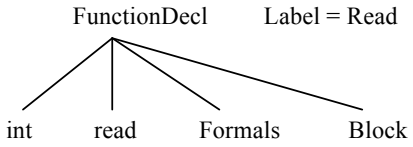
   Pop all local variables; pop block size stack; decrease frame size by the block size just popped; generate a POP instruction to pop the block variables off the runtime stack

Source Program	***Frame Stack*** (after statement)	***Block Stack*** (after statement)	***Comment***
program { int i int j	2	2	push i and j
i = 5	2	2	
{ int k	3	2 1	new block pushed
k = 5 j = k	3	2 1	
}	2	2	pop block; decrease frame size
{ int m	3	2 1	
m = 7 i = m	3	2 1	
}	2	2	
}	-	-	exit main program frame

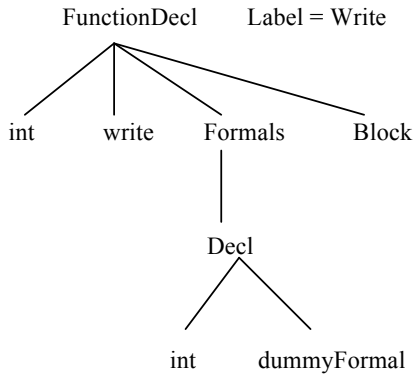
Intrinsic Trees

Generate code for readTree and writeTree; decorate these trees to record the branch labels of these functions for future function calls

readTree:



writeTree:

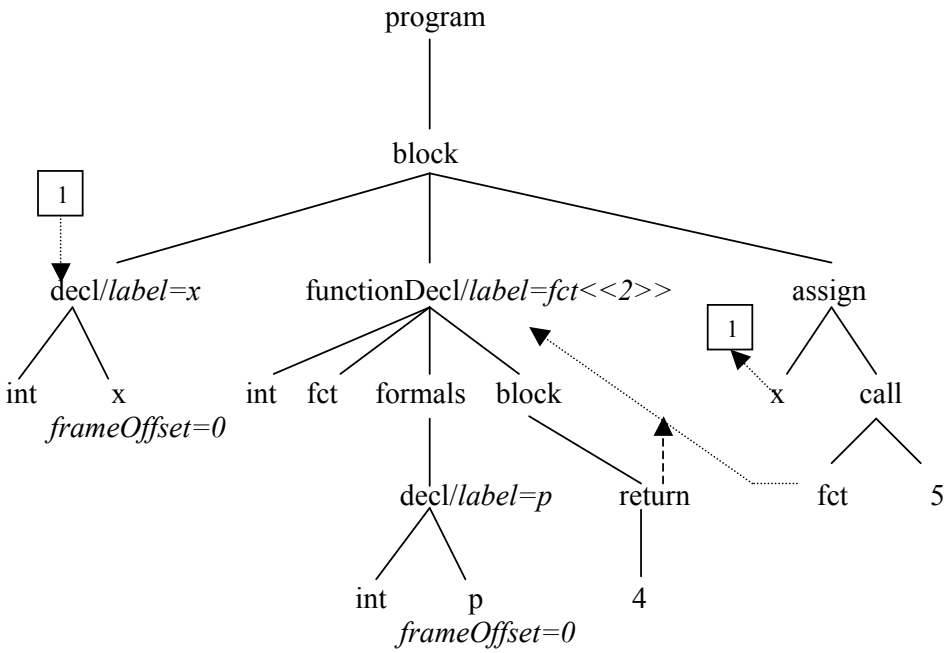


Code generated:

```
LABEL Read
READ
RETURN
LABEL Write
LOAD 0                  -- load argument to be written
WRITE
RETURN
```

Sample Codegen Example

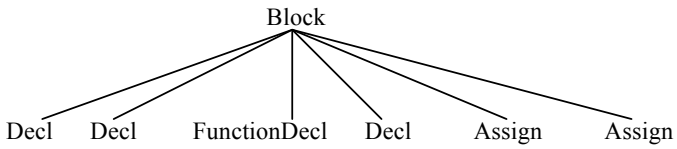
Simulate Codegen with
program { int x
  int fct(int p) { return 4}
  x = fct(5)
}



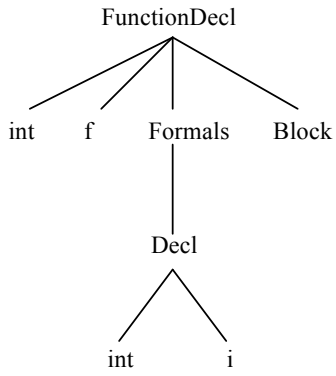

```
GOTO start<<1>>
LABEL Read
READ
RETURN
LABEL Write
LOAD 0 dummyFormal
WRITE
RETURN
LABEL start<<1>>
LIT 0 x
GOTO continue<<3>>
LABEL fct<<2>>
LIT 4
RETURN  fct<<2>>
LIT 0    GRATIS-RETURN-VALUE
RETURN fct<<2>>
LABEL continue<<3>>
LIT 5
ARGS 1
CALL fct<<2>>
STORE 0 x
HALT
```

Processing by Codegen

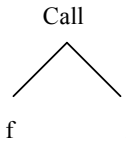

```
 *  GOTO start   -- branch around codes for the intrinsics
 *  <generate codes for the intrinsic trees (read/write)>
 *  LABEL start
 *  <generate codes for the BLOCK tree>
 *  HALT
 *  @param t the program tree to visit
 *  @return null - we're a visitor so must return a value
 *  but the code generator doesn't need any specific value
*/
   public Object visitProgramTree(AST t) {
      String startLabel = newLabel("start");
      openFrame();
      storeop(new LabelOpcode(Codes.ByteCodes.GOTO,startLabel));
      // branch over intrinsic bytecodes
      genIntrinsicCodes();
      storeop(new LabelOpcode(Codes.ByteCodes.LABEL,startLabel));
      t.getKid(1).accept(this);
      storeop(new Code(Codes.ByteCodes.HALT));
      closeFrame();
      return null;
```



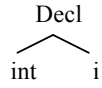

```
   public Object visitBlockTree(AST t) {
      openBlock();
      visitKids(t);
      storeop(new NumOpcode(Codes.ByteCodes.POP,blockSize()));
      // remove any local variables from runtime stack
      closeBlock();
```



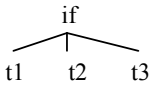

```
*  Generate codes for the function declaration; we'll also record
*  the frame offsets for the formal parameters<br><br>
*  GOTO continue   -- branch around codes for the function
*  LABEL functionLabel
*  <generate codes for the function body>
*  LIT 0
*  RETURN function
*  LABEL continue
   public Object visitFunctionDeclTree(AST t) {
      //System.out.println("visitFunctionDeclTree");
      AST name = t.getKid(2),
         formals = t.getKid(3),
         block = t.getKid(4);
      String funcName = ((IdTree)name).getSymbol().toString();
      String funcLabel = newLabel(funcName);
      t.setLabel(funcLabel);
      String continueLabel = newLabel("continue");
      storeop(new LabelOpcode(Codes.ByteCodes.GOTO,continueLabel));
      openFrame();  // track Frame changes within function
      storeop(new LabelOpcode(Codes.ByteCodes.LABEL,funcLabel));
      // now record the frame offsets for the formals
      for (AST decl : formals.getKids()) {
         IdTree id = (IdTree)(decl.getKid(2));
         id.setFrameOffset(frameSize());
         decl.setLabel(id.getSymbol().toString());
         changeFrame(1);  // ensure frame size includes space for variables
      }
      block.accept(this);
      // emit gratis return in case user didn't provide her/his own return
      storeop(new VarOpcode(Codes.ByteCodes.LIT,0,"   GRATIS-RETURN-
VALUE"));
      storeop(new LabelOpcode(Codes.ByteCodes.RETURN,funcLabel));
      closeFrame();
      storeop(new LabelOpcode(Codes.ByteCodes.LABEL,continueLabel));
      return null;
   }
```



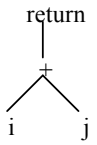

```
*  <Codes to evaluate the arguments for the function>
*  ARGS n    where n is the number of args
*  CALL functionName

  public Object visitCallTree(AST t) {
     //System.out.println("visitCallTree");
     String funcName = ((IdTree)t.getKid(1)).getDecoration().getLabel();
     int numArgs = t.kidCount() - 1;
     for (int kid = 2; kid <= t.kidCount(); kid++) {
        t.getKid(kid).accept(this);
     }
     storeop(new NumOpcode(Codes.ByteCodes.ARGS,numArgs));
     //used to set up new frame
     storeop(new LabelOpcode(Codes.ByteCodes.CALL, funcName));
     return null;
  }
```



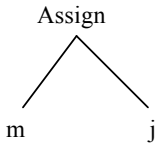

```
*  LIT 0  -- 0 is the initial value for the variable
*  record the frame offset of this variable for future references

  public Object visitDeclTree(AST t) {
     //System.out.println("visitDeclTree");
     IdTree id = (IdTree)t.getKid(2);
     String idLabel = id.getSymbol().toString();
     t.setLabel(idLabel);  //set label in dcln node
     id.setFrameOffset(frameSize());
     storeop(new VarOpcode(Codes.ByteCodes.LIT,0,idLabel));
     //reserve space in frame for new variable; init to 0
     return null;
  }
```




```
*  <generate codes for the conditional tree - t1>
*  FALSEBRANCH elseLabel
*  <generate codes for the then tree - t2>
*  GOTO continue
*  LABEL elseLabel
*  <generate codes for the else tree - t3>
*  LABEL continue

   public Object visitIfTree(AST t) {
      //System.out.println("visitIfTree");
      String elseLabel = newLabel("else"),
            continueLabel = newLabel("continue"),
            nextLabel;
      t.getKid(1).accept(this); // gen code for conditional expr
      nextLabel = elseLabel;
      storeop(new LabelOpcode(Codes.ByteCodes.FALSEBRANCH,nextLabel));
      t.getKid(2).accept(this);
      storeop(new LabelOpcode(Codes.ByteCodes.GOTO,continueLabel));
      storeop(new LabelOpcode(Codes.ByteCodes.LABEL,elseLabel));
      t.getKid(3).accept(this);
      storeop(new LabelOpcode(Codes.ByteCodes.LABEL,continueLabel));
      return null; }
```




```
*  <generate codes for the expression that will be returned>
*  RETURN <name-of-function>

   public Object visitReturnTree(AST t) {
      //System.out.println("visitReturnTree");
      t.getKid(1).accept(this);
      AST fct = t.getDecoration();
      storeop(new LabelOpcode(Codes.ByteCodes.RETURN,fct.getLabel()));
      return null;
   }
```



* <generate codes for the right-hand-side expression>
* STORE *offset-of-variable name-of-variable*

```
public Object visitAssignTree(AST t) {
    //System.out.println("visitAssignTree");
    IdTree id = (IdTree)t.getKid(1);
    String vname = id.getSymbol().toString();
    int addr = ((IdTree)(id.getDecoration().getKid(2))).getFrameOffset();
    t.getKid(2).accept(this);
    storeop(new VarOpcode(Codes.ByteCodes.STORE,addr,vname));
    return null;
}
```

Codegen.java

```
package codegen;

import constrain.*;
import visitor.*;
import java.util.*;
import ast.*;

/**
 *  The Frame class is used for tracking the frame size as we generated
 *  bytecodes; we need the frame size to determine offsets for variables
*/
class Frame {
   private int size = 0;     // size of current frame
   private Stack<Block> blockSizes = new Stack<Block>();
      // If we are embedded 3 blocks deep in the current frame
      // then the blockSizes stack will have 3 items in it,
      // each recording the size of the associated block
      // e.g. consider the following source program segment
      //     int f(int n,int p) {  <- bottom block has formals
      //        int i;             <- next block has i
      //        { int k; int l; int m <- next block has k, l, and m
      //         ...
      //        the blockSizes stack has 2,1,3 with 3 at the top
      //        the framesize is 6 - the sum of all the block sizes

   public Frame() {
      openBlock();
   }

   int getSize() {
      return size;
   }

   void openBlock() {
      blockSizes.push(new Block());
   }

   int closeBlock() {
      int bsize = getBlockSize();
      size -= bsize; // all items in the current block are gone
                     // so decrease the frame size
      blockSizes.pop();
      return bsize;
   }


   Block topBlock() {
      return (Block)blockSizes.peek();
   }

   void change(int n) {   // change current block size; framesize
      size += n;
```

```
      topBlock().change(n);
   }

   int getBlockSize() {
      return topBlock().getSize();
   }
}

/**
 *  The Block class is used to record the size of the current block
 *  Used in tandem with Frame
*/
class Block {
   int size = 0;

   void change(int n) {
      size += n;
   }

   int getSize() {
      return size;
   }
}

/**
 *  The Codegen class will walk the AST, determine and set variable
 *  offsets and generate the bytecodes
*/
public class Codegen extends ASTVisitor {

   AST t;
   Stack<Frame> frameSizes;   // used for tracking the frame sizes;
                  // when we start generating code for a
                  // function we'll push a new entry on the
                  // stack with init size zero

   Program program;    // program will contain the generated bytecodes
   int labelNum;       // used for creating new, unique labels

/**
 *  Create a new code generator based on the given AST
 *  @param t is the AST that will be visited
*/
   public Codegen(AST t) {
      this.t = t;
      program = new Program();
      frameSizes = new Stack<Frame>();
      labelNum = 0;
   }

/** visit all the nodes in the AST/gen bytecodes
*/
   public Program execute() {
      t.accept(this);  //
      return program;
   }
```

```
   Frame topFrame() {
      if (frameSizes.empty())
      System.out.println("frames empty");
      return (Frame)frameSizes.peek();
   }

   void openFrame() {  // open a new frame - we're generating codes for
                       // a function declaration
      frameSizes.push(new Frame());
   }

   void openBlock() {  // open a new block - store local variables
      topFrame().openBlock();
   }

   void closeBlock() {  // close the current block (and pop the local
                        // variables off the runtime stack
      topFrame().closeBlock();
   }

   void closeFrame() {
      frameSizes.pop();
   }

   void changeFrame(int n) { // change the size of the current frame by the
                             // given amount
      topFrame().change(n);
   }

   int frameSize() {  // return the current frame size
      return topFrame().getSize();
   }

   int getBlockSize() {
      return topFrame().getBlockSize();
   }

/** <pre>
 *  we'll need to create new labels for the bytecode program
 *  e.g. the following is legal despite 2 functions with the same name
 *       int f(int n) {... {int f() {... x = f()} } ... y = f(5)}
 *  we don't want to generate the label f for the start of both functions
 *  instead, we'll generate, e.g., f<<0>> and f<<1>>
 *  </pre>
*/
   String newLabel(String label) {  // create a new label from label
      ++labelNum;
      return label + "<<" + labelNum + ">>";
   }

   void storeop(Code code) {
/*
      System.out.println("storeop: "+code.toString()+" fs:"+
        top.getSize()+" bs: "+top.getBlockSize());
```

```
*/
      Codes.ByteCodes bytecode = code.getBytecode();
      int change = Codes.frameChange.get(bytecode);
      program.storeop(code);
      if (change == Codes.UnknownChange) {  // pop n; args n
         changeFrame( - ((NumOpcode)code).getNum());
      } else {
         changeFrame(change);
      }
   }

   void genIntrinsicCodes() {
      // generate codes for read/write functions so they're treated
      // as any other function
      String readLabel = "Read",
           writeLabel = "Write";
      AST readTree = Constrainer.readTree,
         writeTree = Constrainer.writeTree;
      readTree.setLabel(readLabel);
      storeop(new LabelOpcode(Codes.ByteCodes.LABEL,readLabel));
      storeop(new Code(Codes.ByteCodes.READ));
      storeop(new Code(Codes.ByteCodes.RETURN));

      writeTree.setLabel(writeLabel);
      storeop(new LabelOpcode(Codes.ByteCodes.LABEL,writeLabel));
      String formal = ((IdTree)(writeTree.getKid(3).getKid(1).getKid(2))).
                        getSymbol().toString();
      storeop(new VarOpcode(Codes.ByteCodes.LOAD,0,formal));
      // write has one actual arg - in frame offset 0
      storeop(new Code(Codes.ByteCodes.WRITE));
      storeop(new Code(Codes.ByteCodes.RETURN));
   }

/** <pre>
 *  visit the given program AST:<br><br>
 *
 *  GOTO start   -- branch around codes for the intrinsics
 *  &LT;generate codes for the intrinsic trees (read/write)&GT;
 *  LABEL start
 *  &LT;generate codes for the BLOCK tree&GT;
 *  HALT
 *  </pre>
 *  @param t the program tree to visit
 *  @return null - we're a visitor so must return a value
 *  but the code generator doesn't need any specific value
*/
   public Object visitProgramTree(AST t) {
      String startLabel = newLabel("start");
      openFrame();
      storeop(new LabelOpcode(Codes.ByteCodes.GOTO,startLabel));
      // branch over intrinsic bytecodes
      genIntrinsicCodes();
      storeop(new LabelOpcode(Codes.ByteCodes.LABEL,startLabel));
      t.getKid(1).accept(this);
      storeop(new Code(Codes.ByteCodes.HALT));
      closeFrame();
```

```
      return null;
   }

/** <pre>
 *  Generate codes for the Block tree:<br><br>
 *  &LT;codes for the decls and the statements in the block&GT;
 *  POP n   -- n is the number of local variables; pop them
 *  </pre>
*/
   public Object visitBlockTree(AST t) {
      //System.out.println("visitBlockTree");
      openBlock();
      visitKids(t);
      storeop(new NumOpcode(Codes.ByteCodes.POP,getBlockSize()));
      // remove any local variables from runtime stack
      closeBlock();
      return null; }

/** <pre>
 *  Generate codes for the function declaration; we'll also record
 *  the frame offsets for the formal parameters<br><br>
 *  GOTO continue   -- branch around codes for the function
 *  LABEL functionLabel
 *  &LT;generate codes for the function body&GT;
 *  LIT 0
 *  RETURN function
 *  LABEL continue
 *  </pre>
*/
   public Object visitFunctionDeclTree(AST t) {
      //System.out.println("visitFunctionDeclTree");
      AST name = t.getKid(2),
          formals = t.getKid(3),
          block = t.getKid(4);
      String funcName = ((IdTree)name).getSymbol().toString();
      String funcLabel = newLabel(funcName);
      t.setLabel(funcLabel);
      String continueLabel = newLabel("continue");
      storeop(new LabelOpcode(Codes.ByteCodes.GOTO,continueLabel));
      openFrame();  // track Frame changes within function
      storeop(new LabelOpcode(Codes.ByteCodes.LABEL,funcLabel));
      // now record the frame offsets for the formals
      for (AST decl : formals.getKids()) {
         IdTree id = (IdTree)(decl.getKid(2));
         id.setFrameOffset(frameSize());
         decl.setLabel(id.getSymbol().toString());
         changeFrame(1);  // ensure frame size includes space for variables
      }
      block.accept(this);
      // emit gratis return in case user didn't provide her/his own return
      storeop(new VarOpcode(Codes.ByteCodes.LIT,0,"   GRATIS-RETURN-
VALUE"));
      storeop(new LabelOpcode(Codes.ByteCodes.RETURN,funcLabel));
      closeFrame();
      storeop(new LabelOpcode(Codes.ByteCodes.LABEL,continueLabel));
      return null;
```

```
   }

/** <pre>
 *  Generate codes for the call tree:<br><br>
 *
 *  &LT;Codes to evaluate the arguments for the function&GT;
 *  ARGS <i>n</i>    where <i>n</i> is the number of args
 *  CALL functionName
 *  </pre>
*/
   public Object visitCallTree(AST t) {
      //System.out.println("visitCallTree");
      String funcName = ((IdTree)t.getKid(1)).getDecoration().getLabel();
      int numArgs = t.kidCount() - 1;
      for (int kid = 2; kid <= t.kidCount(); kid++) {
         t.getKid(kid).accept(this);
      }
      storeop(new NumOpcode(Codes.ByteCodes.ARGS,numArgs));
      //used to set up new frame
      storeop(new LabelOpcode(Codes.ByteCodes.CALL, funcName));
      return null;
   }

/** <pre>
 *  Generate codes for the Decl tree:<br><br>
 *
 *  LIT 0  -- 0 is the initial value for the variable
 *  record the frame offset of this variable for future references
 *  </pre>
*/
   public Object visitDeclTree(AST t) {
      //System.out.println("visitDeclTree");
      IdTree id = (IdTree)t.getKid(2);
      String idLabel = id.getSymbol().toString();
      t.setLabel(idLabel);  //set label in dcln node
      id.setFrameOffset(frameSize());
      storeop(new VarOpcode(Codes.ByteCodes.LIT,0,idLabel));
      //reserve space in frame for new variable; init to 0
      return null;
   }

   public Object visitIntTypeTree(AST t) {
      //System.out.println("visitIntTypeTree");
      return null; }

   public Object visitBoolTypeTree(AST t) {
      //System.out.println("visitBoolTypeTree");
      return null; }

   public Object visitFormalsTree(AST t) {
      //System.out.println("visitFormalsTree");
      return null; }

   public Object visitActualArgsTree(AST t) {
      //System.out.println("visitActualArgsTree");
      return null; }
```

```
/** <pre>
 *  Generate codes for the <i>If</i> tree:<br><br>
 *
 *  &LT;generate codes for the conditional tree&GT;
 *  FALSEBRANCH elseLabel
 *  &LT;generate codes for the <i>then</i> tree - 2nd kid&GT;
 *  GOTO continue
 *  LABEL elseLabel
 *  &LT;generate codes for the <i>else</i> tree - 3rd kid&GT;
 *  LABEL continue
 *  </pre>
*/
   public Object visitIfTree(AST t) {
      //System.out.println("visitIfTree");
      String elseLabel = newLabel("else"),
            continueLabel = newLabel("continue");
      t.getKid(1).accept(this); // gen code for conditional expr
      storeop(new LabelOpcode(Codes.ByteCodes.FALSEBRANCH, elseLabel));
      t.getKid(2).accept(this);
      storeop(new LabelOpcode(Codes.ByteCodes.GOTO,continueLabel));
      storeop(new LabelOpcode(Codes.ByteCodes.LABEL,elseLabel));
      t.getKid(3).accept(this);
      storeop(new LabelOpcode(Codes.ByteCodes.LABEL,continueLabel));
      return null; }

/** <pre>
 *  Generate codes for the While tree:<br><br>
 *
 *  LABEL while
 *  &LT;generate codes for the conditional&GT;
 *  FALSEBRANCH continue
 *  &LT;generate codes for the body of the while&GT;
 *  GOTO while
 *  LABEL continue
 *  </pre>
*/
   public Object visitWhileTree(AST t) {
      //System.out.println("visitWhileTree");
      String continueLabel = newLabel("continue"),
            whileLabel = newLabel("while");
      storeop(new LabelOpcode(Codes.ByteCodes.LABEL,whileLabel));
      t.getKid(1).accept(this);
      storeop(new LabelOpcode(Codes.ByteCodes.FALSEBRANCH,continueLabel));
      t.getKid(2).accept(this);
      storeop(new LabelOpcode(Codes.ByteCodes.GOTO,whileLabel));
      storeop(new LabelOpcode(Codes.ByteCodes.LABEL,continueLabel));
      return null;
   }

/** <pre>
 *  Generate codes for the return tree:<br><br>
 *
 *  &LT;generate codes for the expression that will be returned&GT;
 *  RETURN &LT;name-of-function&GT;
 *  </pre>
```

```
*/
   public Object visitReturnTree(AST t) {
      //System.out.println("visitReturnTree");
      t.getKid(1).accept(this);
      AST fct = t.getDecoration();
      storeop(new LabelOpcode(Codes.ByteCodes.RETURN,fct.getLabel()));
      return null;
   }

/** <pre>
 *  Generate codes for the Assign tree:<br><br>
 *
 *  &LT;generate codes for the right-hand-side expression&GT;
 *  STORE <i>offset-of-variable name-of-variable</i>
 *  </pre>
*/
   public Object visitAssignTree(AST t) {
      //System.out.println("visitAssignTree");
      IdTree id = (IdTree)t.getKid(1);
      String vname = id.getSymbol().toString();
      int addr = ((IdTree)(id.getDecoration().getKid(2))).getFrameOffset();
      t.getKid(2).accept(this);
      storeop(new VarOpcode(Codes.ByteCodes.STORE,addr,vname));
      return null;
   }

/** <pre>
 *  Load a literal value:
 *  LIT <i>n</i>  n is the value
 *  </pre>
*/
   public Object visitIntTree(AST t) {
      //System.out.println("visitIntTree");
      int num = Integer.parseInt(
                    ((IntTree)t).getSymbol().toString());
      storeop(new NumOpcode(Codes.ByteCodes.LIT,num));
      return null;
   }

/** <pre>
 *  Load a variable:
 *  LOAD <i>offset</i>  -- load variable using the offset recorded in the AST
 *  </pre>
*/
   public Object visitIdTree(AST t) {
      //System.out.println("visitIdTree");
      AST decl = t.getDecoration();
      int addr = ((IdTree)(decl.getKid(2))).getFrameOffset();
      String vname = ((IdTree)t).getSymbol().toString();
      storeop(new VarOpcode(Codes.ByteCodes.LOAD,addr,vname));
      return null;
   }

/** <pre>
 *  Generate codes for the relational op tree e.g. t1 == t2<br><br>
 *
```

```
 *  &LT;generate codes for t1&GT;
 *  &LT;generate codes for t2&GT;
 *  BOP op    -- op is the indicated relational op
 *  </pre>
*/
   public Object visitRelOpTree(AST t) {
      //System.out.println("visitRelOpTree");
      String op = ((RelOpTree)t).getSymbol().toString();
      t.getKid(1).accept(this);
      t.getKid(2).accept(this);
      storeop(new LabelOpcode(Codes.ByteCodes.BOP,op));
      return null;
   }

/** <pre>
 *  Generate codes for the adding op tree e.g. t1 + t2<br><br>
 *
 *  &LT;generate codes for t1&GT;
 *  &LT;generate codes for t2&GT;
 *  BOP op    -- op is the indicated adding op
 *  </pre>
*/
   public Object visitAddOpTree(AST t) {
      //System.out.println("visitAddOpTree");
      String op = ((AddOpTree)t).getSymbol().toString();
      t.getKid(1).accept(this);
      t.getKid(2).accept(this);
      storeop(new LabelOpcode(Codes.ByteCodes.BOP,op));
      return null;
   }

/** <pre>
 *  Generate codes for the multiplying op tree e.g. t1 * t2<br><br>
 *
 *  &LT;generate codes for t1&GT;
 *  &LT;generate codes for t2&GT;
 *  BOP op    -- op is the indicated multiplying op
 *  </pre>
*/
   public Object visitMultOpTree(AST t) {
      //System.out.println("visitMultOpTree");
      String op = ((MultOpTree)t).getSymbol().toString();
      t.getKid(1).accept(this);
      t.getKid(2).accept(this);
      storeop(new LabelOpcode(Codes.ByteCodes.BOP,op));
      return null;
   }
}
```

XVI. Chapter 10 - Mechanisms for Software Reuse

Inheritance vs. Composition (aggregation) - 1

(is-a vs. has-a)

Car is-a Vehicle (inheritance)

Car has-a Engine (composition)

Will the is-a relationship be ***constant*** throughout the lifetime of the application?

e.g.

Initially, *Employee is-a Person*; but, Employee ***might be a role*** played by a Person at certain points in time; what if the Person becomes unemployed? What if the Person is both an Employee and a Supervisor?

Normally, we use Composition to model these relationships

```
class Person {
    Role workingRole;
    ...
}
```

Abstract classes vs. Interfaces

Abstract:

use when you want to ***share the structure or code*** among super- and sub-classes

Interfaces:

Use when you want to ***share the specification of behavior*** but no code

```
public abstract class AST {
   protected ArrayList<AST> kids;
   ...
   public void setDecoration(AST t) {
       // note the naming of this accessor - setter
      decoration = t;
   }
...
}

public interface java.util.Enumeration {
       public abstract boolean hasMoreElements();
       public abstract Object nextElement();
}
```

Inheritance vs. Composition (aggregation) - 2

Consider Stacks in Java

```
class Stack extends Vector {
        public Object push(Object item) { addElement(item); return item; }
        public Object peek() {return elementAt(size() - 1); }
        ...
}
```

- Stacks can use *protected* members of Vector
- Stacks can be used wherever Vectors are required as *arguments*
- Reuse Vector methods such as *size*, *isEmpty*
- Reuse methods that might not make sense for Stacks (we can remove any item in a Vector, not just the "topmost")

```
class Stack {

        private Vector theData;

        public Stack() { theData = new Vector(); }
        public boolean isEmpty() { return theData.isEmpty(); }

        ...

}
```

- ***We can change the implementation*** of Stack from Vectors to, e.g., arrays ***without any impact*** on code that uses Stacks
- ***Surface area is smaller*** (don't need to understand Vectors in order to understand Stacks)
- ***Meaningless behavior isn't exposed*** (by extending Vector we can do things to Stacks like insertElementAt())

1. **Substitutability**: inheritance has this, not composition
2. Composition: easier to understand operations that are available (see the Stack example above - insertElementAt())
3. Inheritance: ***tight coupling*** of subclass with superclasses
4. ***Changes*** to superclass have ***rippling effect*** on all subclasses
5. Users of class with inheritance must look at the class and all superclasses to understand behavior - ***large surface area***
6. Implementation is shorter using inheritance due to code reuse
7. Inheritance: might use methods in superclass that are not really appropriate
8. Composition: changes to private data have no effect on users of class
9. Field/Method access: with inheritance superclass members are accessible if they're declared ***public or protected***; whereas with composition only ***public*** members are available in the composed class
10. Inheritance: polymorphism is available - consider the use of AST's in the compiler (wherever we expect an AST we can supply an instance of any subclass of AST); with composition polymorphism is not available

Combining Composition and Inheritance

- Very powerful
- Can be used to greatly simplify the inheritance hierarchy

Consider the Java Stream Wrappers:

```
InputStream
        ByteArrayInputStream
        FileInputStream
        PipedInputStream
        SequenceInputStream
        ObjectInputStream
        FilterInputStream
            BufferedInputStream
            DataInputStream
```

Subclassing allows processing an InputStream regardless of where bytes come from;

```
class FilterInputStream extends InputStream {
        ...
        protected InputStream in;    // source of bytes is from any InputStream
        FilterInputStream(InputStream in) {
            this.in = in;
        }
        ...
}
```

FilterInputStreams ***can be used wherever we expect an InputStream*** since it subclasses

InputStream

It gets the bytes from *in* and *then performs filtering*

In this case, we are provided with ***orthogonal sources of variation***; the source of bytes

and added functionality (e.g. filtering, buffering); ***we are avoiding an explosion of the***

inheritance hierarchy

If we didn't use composition with inheritance:

```
InputStream
        FilterInputStream
            BufferedInputStream
                    BufferedDataInputStream
                            -- lots of these classes would be needed
            DataInputStream


Reader
        BufferedReader
```

```
package java.io;

/**
 * Read text from a character-input stream, buffering characters so as to
 * provide for the efficient reading of characters, arrays, and lines.
 *
 * In general, each read request made of a Reader causes a corresponding
 * read request to be made of the underlying character or byte stream.  It is
 * therefore advisable to wrap a BufferedReader around any Reader whose read()
 * operations may be costly, such as FileReaders and InputStreamReaders.  For
 * example,
 *
 * BufferedReader in
 *   = new BufferedReader(new FileReader("foo.in"));
 *
 * will buffer the input from the specified file.  Without buffering, each
 * invocation of read() or readLine() could cause bytes to be read from the
 * file, converted into characters, and then returned, which can be very
 * inefficient.
 *

public class BufferedReader extends Reader {

   private Reader in;
        // composition adds buffering services to the Reader it's wrapping
   private char cb[];       // the buffered characters read from in

   private static int defaultCharBufferSize = 8192;
   private static int defaultExpectedLineLength = 80;

   /**
    * Create a buffering character-input stream that uses an input buffer of
    * the specified size.
    *
    * @param  in   A Reader
    * @param  sz   Input-buffer size
    *
    * @exception  IllegalArgumentException  If sz is <= 0
    */
   public BufferedReader(Reader in, int sz) {
       super(in);
       if (sz <= 0)
           throw new IllegalArgumentException("Buffer size <= 0");
       this.in = in;
       cb = new char[sz];
       nextChar = nChars = 0;
   }

   /**
    * Create a buffering character-input stream that uses a default-sized
    * input buffer.
    *
    * @param  in   A Reader
    */
```

```
public BufferedReader(Reader in) {
    this(in, defaultCharBufferSize);  // use other constructor
}


/**
 * Read a single character.
 *
 * @exception  IOException  If an I/O error occurs
 */
public int read() throws IOException {
}


/**
 * Read a line of text.  A line is considered to be terminated by any one
 * of a line feed ('\n'), a carriage return ('\r'), or a carriage return
 * followed immediately by a linefeed.
 *
 * @return     A String containing the contents of the line, not including
 *             any line-termination characters, or null if the end of the
 *             stream has been reached
 *
 * @exception  IOException  If an I/O error occurs
 */
public String readLine() throws IOException {
```

Dynamic Composition

```
class Frog {
        private FrogBehavior behavior;

        public Frog() {
            behavior = new TadpoleBehavior();
        }

        public grow() { // see if behavior should change
            if (behavior.growUp())
                        behavior = new AdultFrogBehavior();
            behavior.grow();   // behavior does actual work
            behavior.swim();
        }
}
```

Note that we can change the behavior as the program is running (***dynamically***)
We cannot change a superclass

```
abstract class FrogBehavior {
        public boolean growUp() { return false; }
        public abstract void grow();
        public abstract swim();
}

class TadpoleBehavior extends FrogBehavior {
        private int age = 0;
        public boolean growUp() { return (++age > 24)? true : false; }
        public void grow() { ... }
        public void swim() {...}
}

class AdultFrogBehavior extends FrogBehavior {
        public void grow() { ... }
        public void swim() { ... }
}
```

Reading:

Budd text: Chapter 10

XVII. Chapter 11 - Implications of Inheritance

- The language should support the ***polymorphic variable***:

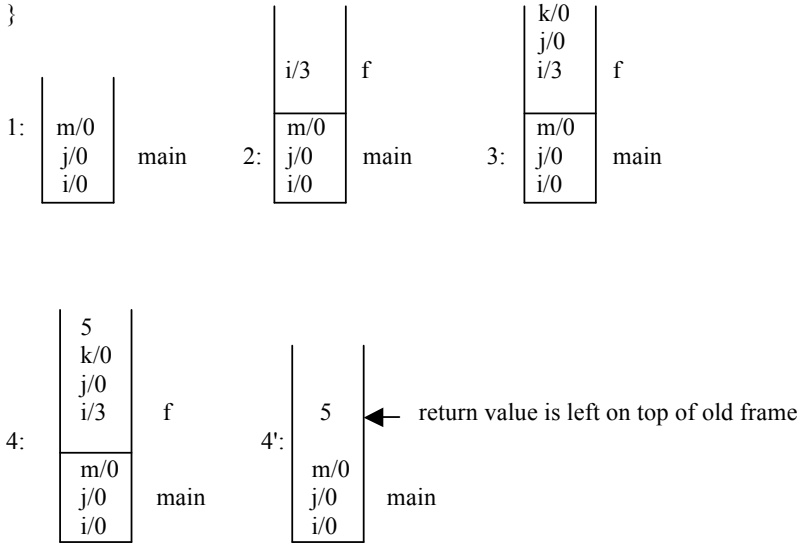
  AST t -- t can hold *any* subclass of AST
- When using polymorphic variables we need to allocate the memory for objects on the ***heap***
- Since the variables reside on the heap it's natural to use ***reference semantics*** for assignment (variables reference objects; if we assign x = y then both x and y reference the same object)
- We need to consider both strong and weak equality: ***Strong equality*** is object identity (x == y is true precisely when both x and y reference the same object); ***Weak equality*** is left to the programmer to define (e.g. two people might be equal if they both have the same name)
- Since the heap is used for memory we need a memory management system (***garbage collection***)

Polymorphic Variables

```
public AST rProgram() throws SyntaxError {
   AST t = new ProgramTree();
   expect(Tokens.Program);
   t.addKid(rBlock());
   return t;
}

public Compiler(String sourceFile) {
      ...
      AST t = parser.execute();
      PrintVisitor pv = new PrintVisitor();
      t.accept(pv);
      ...
}
```

Memory Layout

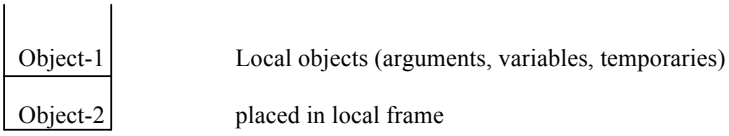
Recall the runtime stack of our machine:



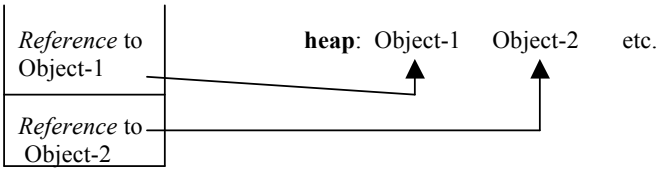
We can compute offsets since we know that each variable is an *int* or *boolean* which occupies one slot; however, if we place AST's on the stack then we cannot predict how much space to allocate (we don't know which subclass object will be placed on the stack

beforehand). Since addresses take a fixed amount of space if we put references to the objects on the stack then we can predict how much stack space to allocate.

Instead of:



We can use:



The ***heap*** is a general storage area used for ***non-stack memory requests*** - uses storage management system to track free store (***garbage collection***)

Assignment

```
public class Box {
        private int value;
        public Box() { value = 0; }
        public void setValue(int v) { value = v; }
        public int getValue() { return value; }
}


public class BoxTest {
        public static void main(String args[ ]) {
            Box x = new Box();
            x.setValue(7);

            Box y = x;
            // reference semantics is used - both x and y refer to the same Box object
            y.setValue(11);

            System.out.println(x.getValue());       // 11 is printed
            System.out.println(y.getValue());       // 11 is printed
        }
}
```

Clones

```
public class Box {
        private int value;
        public Box() { value = 0; }
        public void setValue(int v) { value = v; }
        public int getValue() { return value; }
        public Box copy() {
            Box b = new Box();       b.setValue(value);     return b}
}


public class BoxTest {
        public static void main(String args[ ]) {
            Box x = new Box();
            x.setValue(7);

            Box y = x.copy();
            y.setValue(11);

            System.out.println(x.getValue());       // 7 is printed
            System.out.println(y.getValue());       // 11 is printed
        }
}
```

Box using Object *clone*

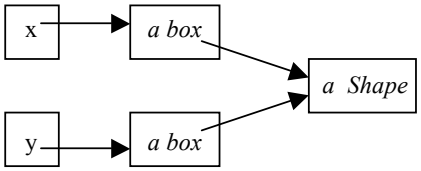
The Object class provides a method *clone()* that makes a ***bitwise copy*** of the

Object. If a bitwise copy isn't desired then the class can implement its own *clone*

```
public class Box implements Cloneable {
        private int value;
        public Box() { value = 0; }
        public void setValue(int v) { value = v; }
        public int getValue() { return value; }
        public Object clone() {
           try {
              return super.clone();     // clone is a protected method in Object
           } catch (CloneNotSupportedException e) { return null; }
        }
}


public class BoxTest {
        public static void main(String args[ ]) {
            Box x = new Box();
            x.setValue(7);

            Box y = (Box)x.clone();
            y.setValue(11);

            System.out.println(x.getValue());       // 7 is printed
            System.out.println(y.getValue());       // 11 is printed
        }
}
```

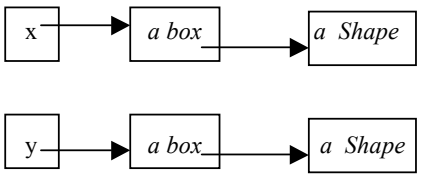
Shallow vs. Deep Copies

Variations on performing a clone where Boxes have a field that references *Shape*:



Shallow Copy

:



Deep Copy

Note: Vector is Cloneable: Vector v1 = v.clone() -- used extensively in *threaded applications*.

Parameters as a Form of Assignment

```
public class BoxTest {
        public static void main (String args[ ]) {
            Box x = new Box();
            x.setValue(7);

            sneaky(x);
            System.out.println(x.getValue());     // 11 is printed
        }
        static void sneaky(Box y) {
        // y is assigned x      -- y references the same Box object
            y.setValue(11);
        }
}
```

Equality

Strong Equality - object identity

AST t = new ProgramTree();

...

AST tt = new ProgramTree();

...

AST ttt = t;

if (t == tt) // true or false?

if (t == ttt) // true or false?

if (t == null) // does *t* reference a null value?

Weak Equality - programmers must define an *equals* method

String s = "abc"

...

if (s.equals("abc")) ...

Garbage Collection

There are too many problems when each user manages their own free storage! *(C++)*

AST t = new ProgramTree();

...

t = null;

// now the *ProgramTree* object has no references to it - it's *garbage so its storage can be reclaimed*

A main issue in garbage collection is the *time overhead*.

Reading:

Budd text: Chapter 11

XVIII. Chapter 12 - Polymorphism

Polymorphic Variables: variables declared as one type but hold values of another type

AST t;

- *t* holds objects whose classes are *subclasses* of AST
- Substitutability applies

Overloading

Several functions with the same name - arguments determine which body to use

e.g.

```
class XYZ {
        int print() {
            ...
        }
        int print(float f) {        PARAMETRIC OVERLOADING
            ..
        }
}
```

Used for providing *consistency and uniformity* - in each case above we want to *print*

More examples:

1 + 2 => integer addition

1.2 + 2.3 => float addition

```
Vector                  HashTable
    isEmpty()               isEmpty()
```

Overriding

Methods with the *same type signatures* that appear in a *super and*

subclass; if the method is public (private) in the superclass then it must

be public (private) in the subclass.

e.g.

Object
toString()

```
java.lang.Object
    |
    +----lexer.Symbol
```

public class **Symbol**

extends Object The Symbol class is used to store all user strings along with an indication of the kind of strings they are; e.g. the id "abc" will store the "abc" in name and Identifier in kind

toString

```
public String toString()
```

Overrides:

toString in class Object

Replacement and Refinement

toString() in Symbol **replaces** *toString* in *Object*

Constructors in Java use ***refinement***

Subclass construction ***always*** calls superclass constructor

Consider the modification of toString in Symbol.java:

```
public String toString() {
    return name + super.toString();
}
```

This *refinement* version ***adds additional behavior*** to the superclass *toString*

Abstract Methods - specify that the behavior is deferred

Announces that the behavior must be defined in subclass(es) that will build instances

(non-abstract subclasses)

```
public abstract class AST {
   protected Vector kids;
   protected int nodeNum;
   protected AST decoration;
   ...
 *  accept the visitor for this node - this method must be defined in each of
 *  the subclasses of AST
   public abstract Object accept(ASTVisitor v);
```

Efficiency and Polymorphism

The use of ***dynamic binding*** implies there will be ***runtime computation needed*** to

determine the required method body - ***added time***

C++ defaults to non-dynamic binding; *virtual* is used to indicate dynamic binding

should be used

Reading:

Budd text: Chapter 12

XIX. Chapter 14 - Input and Output Streams - Effective Uses of Inheritance with Composition

The design of the Input/Output stream library in Java provides an elegantly designed hierarchy that makes efficient and effective use of

- **Inheritance**
- **Composition**
- **Polymorphism**

There are:

- 8 bit streams (*byte oriented* streams)

  InputStream

  OutputStream

- 16 bit streams (*Unicode character oriented* streams)

  Reader

  Writer

Recall the following:

```
TokenSetup() {
   try {
      in = new BufferedReader( new FileReader("lexer\\setup\\tokens"));
      table = new PrintWriter(new FileOutputStream("lexer\\TokenType.java"));
      symbols = new PrintWriter(new FileOutputStream("lexer\\java"));
   } catch (Exception e) {}
}
```

FileReader - reads characters from file and converts them to UNICODE

BufferedReader - used to ***wrap*** any Reader with buffering capabilities

Readers

public class InputStreamReader extends Reader

java.lang.Object
  |
  +----java.io.Reader
        |
        +----java.io.InputStreamReader
              |
              +----java.io.FileReader

An InputStreamReader is a ***bridge from byte streams to character streams***: It reads bytes and translates them into characters according to a specified character encoding. Each invocation of one of an InputStreamReader's read() methods may cause one or more bytes to be read from the underlying byte-input stream. For top efficiency, consider wrapping an InputStreamReader within a BufferedReader; for example,

BufferedReader in = new BufferedReader(new InputStreamReader(System.in));

InputStreamReader(InputStream) - Create an InputStreamReader that uses the default character encoding.

close() - Close the stream.

read() - Read a single character.

read(char[], int, int) - Read characters into a portion of an array.

ready() - Tell whether this stream is ready to be read.

---

public class BufferedReader extends Reader

java.lang.Object
  |
  +----java.io.Reader
        |
        +----java.io.BufferedReader

Read text from a character-input stream, ***buffering characters so as to provide for the efficient reading of characters, arrays, and lines***. The buffer size may be specified, or the default size may be used. The default is large enough for most purposes. In general, ***each read request made of a Reader causes a corresponding read request to be made of the underlying character or***

byte stream. It is therefore advisable to wrap a BufferedReader around any Reader whose read() operations may be costly, such as FileReaders and InputStreamReaders. For example,

BufferedReader in = new BufferedReader(new FileReader("foo.in"));

will buffer the input from the specified file. Without buffering, each invocation of read() or readLine() could cause bytes to be read from the file, converted into characters, and then returned, which can be very inefficient. Programs that use DataInputStreams for textual input can be localized by replacing each DataInputStream with an appropriate BufferedReader.

BufferedReader(Reader) - Create a buffering character-input stream that uses a default-sized input buffer.

BufferedReader(Reader, int) - Create a buffering character-input stream that uses an input buffer of the specified size.

close() - Close the stream.

read() - Read a single character.

read(char[], int, int) - Read characters into a portion of an array.

readLine() - Read a line of text.

ready() - Tell whether this stream is ready to be read.

InputStreams

public class DataInputStream extends FilterInputStream implements DataInput

java.lang.Object
  |
  +----java.io.InputStream
        |
        +----java.io.FilterInputStream
              |
              +----java.io.DataInputStream

A data input stream lets an application ***read primitive Java data types from an underlying input stream in a machine-independent way***. An application uses a data output stream to write data that can later be read by a data input stream.

DataInputStream(InputStream) - Creates a new data input stream to read data from the specified input stream.

read(byte[]) - Reads up to byte.length bytes of data from this data input stream into an array of bytes.

readBoolean() - Reads a boolean from this data input stream.

readByte() - Reads a signed 8-bit value from this data input stream.

readChar() - Reads a Unicode character from this data input stream.

readDouble() - Reads a double from this data input stream.

readFloat() - Reads a float from this data input stream.

readInt() - Reads a signed 32-bit integer from this data input stream.

readLong() - Reads a signed 64-bit integer from this data input stream.

readShort() - Reads a signed 16-bit number from this data input stream.

skipBytes(int) - Skips exactly n bytes of input in the underlying input stream.

public class BufferedInputStream extends FilterInputStream

java.lang.Object
  |
  +----java.io.InputStream
        |
        +----java.io.FilterInputStream
              |
              +----java.io.BufferedInputStream

The class implements a buffered input stream. By setting up such an input stream, an application can read bytes from a stream without necessarily causing a call to the underlying system for each byte read. ***The data is read by blocks into a buffer; subsequent reads can access the data directly from the buffer***.

BufferedInputStream(InputStream) - Creates a new buffered input stream to read data from the specified input stream with a default 512-byte buffer size.

available() - Returns the number of bytes that can be read from this input stream without blocking.

read() - Reads the next byte of data from this buffered input stream.

read(byte[], int, int) - Reads bytes into a portion of an array.

skip(long) - Skips over and discards n bytes of data from the input stream.

public class FileInputStream extends InputStream

java.lang.Object
  |
  +----java.io.InputStream
        |
        +----java.io.FileInputStream

A file input stream is an input stream for reading data from a File or from a FileDescriptor.

available() - Returns the number of bytes that can be read from this file input stream without blocking.

close() - Closes this file input stream and releases any system resources associated with the stream.

read() - Reads a byte of data from this input stream.

read(byte[]) - Reads up to b.length bytes of data from this input stream into an array of bytes.

Note the following use of stream wrappers:

DataInputStream in =

new DataInputStream(

new BufferedInputStream(new FileInputStream("abc")))

try {

int i = in.readInt();

} catch ...

We will get bytes from file "abc"; we'll buffer the bytes and then we'll extract them in terms of ints, floats, etc. **Each wrapper provides additional services**. These wrappers are used ***instead of providing additional services by extending classes*** (subclassing).

InputStream Hierarchy

InputStream	**process 8 bit bytes (Abstract)**
ByteArrayInputStream	***Provide Actual Bytes***
FileInputStream	***Provide Actual Bytes***
PipedInputStream	***Provide Actual Bytes***
ObjectInputStream	**Below are virtual streams**
SequenceInputStream	**<other inputstreams provide bytes>**
FilterInputStream	**< these provide added services>**
DataInputStream	
BufferedInputStream	
LineNumberInputStream	
PushbackInputStream	

Without using composition we would need to provide the following hierarchy:

InputStream
- **FileInputStream**
  - **FileBufferedInputStream**
    - **FileBufferedDataInputStream**
- **ByteArrayInputStream**
  - **FileByteArrayInputStream**
    - **FileBufferedByteArrayInputStream**
      - **...**

- we would need to provide a class for each combination!

The hierarchy would explode in terms of size (complexity)

Reading:

Budd text: Chapter 14

XX. Chapter 16 - Exception Handling in Java

Definition: **An error or other exceptional condition** that arises during program execution. To ***raise*** an exception is to abandon normal program execution so as to draw attention to the fact that the corresponding situation has arisen. Executing some actions, in response to the arising of an exception, is called ***handling*** the exception.

```
public AST execute() throws Exception {
   try {
      return rProgram();
   } catch (SyntaxError e) {          // handle exception
      e.print();
      throw e;           // re-throw the same exception; it's raised again
   } catch (IOException io) {       // handle multiple exceptions
       System.out.println
            ("I/O exception while executing Parser: " + io.toString());
       throw io;
    } catch (Exception ex) {   // catch any other problems
       System.out.println("Unanticipated exception while executing Parser: " +
             ex.toString());
    }
     // control resumes here for exceptions caught and not re-thrown
  }

public AST rProgram() throws SyntaxError {
   AST t = new ProgramTree();
   expect(Program);
   t.addKid(rBlock());      // exception would be propagated
   return t;
}

public AST rBlock() throws SyntaxError {
   expect(LeftBrace);
   AST t = new BlockTree();
    ...
```

Other Exception Information

```
try {
   int a[] = new int[2];
   a[4];
} catch (ArrayIndexOutOfBoundsException e) {
   System.out.println("exception: " + e.getMessage());
   e.printStackTrace();
}
```

Partial Java Exception Hierarchy

```
Throwable
      Error                  // these are "hard failures" => stop processing
         LinkageError
                  IncompatibleClassChangeError
         VirtualMachineError
                  InternalError
                  OutOfMemoryError
                  StackOverflowError
      Exception
         IllegalAccessException
         IOException
                  EOFException
                  FileNotFoundException
                  InerruptedIOException
                  MalformedURLException
         RuntimeException                  // these can happen everywhere so
                  ArithmeticException      // the compiler doesn't insist on a
                  ClassCastException       // try/catch block
                  IndexOutOfBoundsException
                  NullPointerException
                  SecurityException
```

```
class SyntaxError extends Exception {
   private Token tokenFound;
   private int kindExpected;

   public SyntaxError(Token tokenFound, int kindExpected) {
      this.tokenFound = tokenFound;
      this.kindExpected = kindExpected;
   }

   void print() {
      System.out.println("Expected: " +
        TokenType.tokens[kindExpected].toString());
      return;
   }
}
```

Reading:

Budd text: Chapter 16

XXI. Chapter 19 - Collection Classes

The Java Collections Framework (Javadoc introduction)

Interface Hierarchy

- interface java.util.**Collection**
  - interface java.util.**List**
  - interface java.util.**Set**
    - interface java.util.**SortedSet**
- interface java.util.**Comparator**
- interface java.util.**Enumeration**
- interface java.util.**Iterator**
  - interface java.util.**ListIterator**
- interface java.util.**Map**
  - interface java.util.**SortedMap**

public interface **Collection**
The root interface in the *collection hierarchy*. A collection represents a group of objects, known as its *elements*. Some collections allow duplicate elements and others do not. Some are ordered and others unordered. The SDK does not provide any *direct* implementations of this interface: it provides implementations of more specific subinterfaces like Set and List. This interface is typically used to pass collections around and manipulate them where maximum generality is desired.
Bags or *multisets* (unordered collections that may contain duplicate elements) should implement this interface directly.
All general-purpose Collection implementation classes (which typically implement Collection indirectly through one of its subinterfaces) should provide two "standard" constructors: a void (no arguments) constructor, which creates an empty collection, and a constructor with a single argument of type Collection, which creates a new collection with the same elements as its argument. In effect, the latter constructor allows the user to copy any collection, producing an equivalent collection of the desired implementation type. There is no way to enforce this convention (as interfaces cannot contain constructors) but all of the general-purpose Collection implementations in the Java platform libraries comply.
The "destructive" methods contained in this interface, that is, the methods that modify the collection on which they operate, are specified to throw UnsupportedOperationException if this collection does not support the operation. If this is the case, these methods may, but are not required to, throw an UnsupportedOperationException if the invocation would have no effect on the collection. For example, invoking the addAll(Collection) method on an unmodifiable collection may, but is not required to, throw the exception if the collection to be added is empty.
Some collection implementations have restrictions on the elements that they may contain. For example, some implementations prohibit null elements, and some have restrictions on the types of their elements. Attempting to add an ineligible element throws an unchecked exception, typically NullPointerException

or ClassCastException. Attempting to query the presence of an ineligible element may throw an exception, or it may simply return false; some implementations will exhibit the former behavior and some will exhibit the latter. More generally, attempting an operation on an ineligible element whose completion would not result in the insertion of an ineligible element into the collection may throw an exception or it may succeed, at the option of the implementation. Such exceptions are marked as "optional" in the specification for this interface.

Method Summary	
boolean	**add**(Object o) Ensures that this collection contains the specified element (optional operation).
boolean	**addAll**(Collection c) Adds all of the elements in the specified collection to this collection (optional operation).
void	**clear**() Removes all of the elements from this collection (optional operation).
boolean	**contains**(Object o) Returns true if this collection contains the specified element.
boolean	**containsAll**(Collection c) Returns true if this collection contains all of the elements in the specified collection.
boolean	**equals**(Object o) Compares the specified object with this collection for equality.
int	**hashCode**() Returns the hash code value for this collection.
boolean	**isEmpty**() Returns true if this collection contains no elements.
Iterator	**iterator**() Returns an iterator over the elements in this collection.
boolean	**remove**(Object o) Removes a single instance of the specified element from this collection, if it is present (optional operation).
boolean	**removeAll**(Collection c) Removes all this collection's elements that are also contained in the specified collection (optional operation).
boolean	**retainAll**(Collection c) Retains only the elements in this collection that are contained in the specified collection (optional operation).
int	**size**() Returns the number of elements in this collection.
Object[]	**toArray**() Returns an array containing all of the elements in this collection.
Object[]	**toArray**(Object[] a) Returns an array containing all of the elements in this collection; the runtime type of the returned array is that of the specified array.

public interface **List**
extends Collection
An ordered collection (also known as a *sequence*). The user of this interface has precise control over where in the list each element is inserted. The user can access elements by their integer index (position in the list), and search for elements in the list.

public interface **Set**
extends Collection
A collection that contains no duplicate elements. More formally, sets contain no pair of elements `e1` and `e2` such that `e1.equals(e2)`, and at most one null element. As implied by its name, this interface models the mathematical *set* abstraction.

public interface **SortedSet**
extends Set
A set that further guarantees that its iterator will traverse the set in ascending element order, sorted according to the *natural ordering* of its elements (see Comparable), or by a Comparator provided at sorted set creation time. Several additional operations are provided to take advantage of the ordering. (This interface is the set analogue of SortedMap.)

public interface **Comparator**
A comparison function, which imposes a *total ordering* on some collection of objects. Comparators can be passed to a sort method (such as `Collections.sort`) to allow precise control over the sort order. Comparators can also be used to control the order of certain data structures (such as `TreeSet` or `TreeMap`).
The ordering imposed by a Comparator `c` on a set of elements `S` is said to be *consistent with equals* if and only if `(compare((Object)e1, (Object)e2)==0)` has the same boolean value as `e1.equals((Object)e2)` for every `e1` and `e2` in `S`.
Caution should be exercised when using a comparator capable of imposing an ordering inconsistent with equals to order a sorted set (or sorted map). Suppose a sorted set (or sorted map) with an explicit Comparator `c` is used with elements (or keys) drawn from a set `S`. If the ordering imposed by `c` on `S` is inconsistent with equals, the sorted set (or sorted map) will behave "strangely." In particular the sorted set (or sorted map) will violate the general contract for set (or map), which is defined in terms of `equals`.
For example, if one adds two keys `a` and `b` such that `(a.equals((Object)b) && c.compare((Object)a, (Object)b) != 0)` to a sorted set with comparator `c`, the second `add` operation will return false (and the size of the sorted set will not increase) because `a` and `b` are equivalent from the sorted set's perspective.
Note: It is generally a good idea for comparators to implement `java.io.Serializable`, as they may be used as ordering methods in serializable data structures (like `TreeSet`, `TreeMap`). In order for the data structure to serialize successfully, the comparator (if provided) must implement `Serializable`.
For the mathematically inclined, the *relation* that defines the *total order* that a given comparator `c` imposes on a given set of objects `S` is:
    {(x, y) such that c.compare((Object)x, (Object)y) <= 0}.

The *quotient* for this total order is:
    {(x, y) such that c.compare((Object)x, (Object)y) == 0}.

It follows immediately from the contract for `compare` that the quotient is an *equivalence relation* on `S`, and that the natural ordering is a *total order* on `S`. When we say that the ordering imposed by `c` on `S` is *consistent with equals*, we mean that the quotient for the natural ordering is the equivalence relation defined by the objects' `equals(Object)` method(s):
    {(x, y) such that x.equals((Object)y)}.

Method Summary	
int	**compare**(Object o1, Object o2) Compares its two arguments for order. Returns a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second.
boolean	**equals**(Object obj) Indicates whether some other object is "equal to" this Comparator.

public interface **ListIterator**
extends Iterator
An iterator for lists that allows the programmer to traverse the list in either direction, modify the list during iteration, and obtain the iterator's current position in the list.

public interface **Map**
An object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value.
This interface takes the place of the `Dictionary` class, which was a totally abstract class rather than an interface.

public interface **SortedMap**
extends Map
A map that further guarantees that it will be in ascending key order, sorted according to the *natural ordering* of its keys (see the `Comparable` interface), or by a comparator provided at sorted map creation time. This order is reflected when iterating over the sorted map's collection views (returned by the `entrySet`, `keySet` and `values` methods).

Class Hierarchy

- class java.lang.**Object**
  - class java.util.**AbstractCollection** (implements java.util.Collection)
    - class java.util.**AbstractList** (implements java.util.List)
      - class java.util.**AbstractSequentialList**
        - class java.util.**LinkedList** (implements java.lang.Cloneable, java.util.List, java.io.Serializable)
      - class java.util.**Vector** (implements java.lang.Cloneable, java.util.List, java.util.RandomAccess, java.io.Serializable)
        - class java.util.**Stack**
    - class java.util.**AbstractSet** (implements java.util.Set)
      - class java.util.**HashSet** (implements java.lang.Cloneable, java.io.Serializable, java.util.Set)
      - class java.util.**TreeSet** (implements java.lang.Cloneable, java.io.Serializable, java.util.SortedSet)
  - class java.util.**AbstractMap** (implements java.util.Map)
    - public class **HashMap** extends AbstractMap implements Map, Cloneable, Serializable
    - class java.util.**TreeMap** (implements java.lang.Cloneable, java.io.Serializable, java.util.SortedMap)
  - class java.util.**BitSet** (implements java.lang.Cloneable, java.io.Serializable)
  - class java.util.**Dictionary**
    - class java.util.**Hashtable** (implements java.lang.Cloneable, java.util.Map, java.io.Serializable)
      - class java.util.**Properties**

public class **LinkedList**
extends AbstractSequentialList
implements List, Cloneable, Serializable
Linked list implementation of the `List` interface. Implements all optional list operations, and permits all elements (including `null`). In addition to implementing the `List` interface, the `LinkedList` class provides uniformly named methods to `get`, `remove` and `insert` an element at the beginning and end of the list. These operations allow linked lists to be used as a stack, queue, or double-ended queue (deque).

public class **HashSet**
extends AbstractSet
implements Set, Cloneable, Serializable

This class implements the `Set` interface, backed by a hash table (actually a `HashMap` instance). It makes no guarantees as to the iteration order of the set; in particular, it does not guarantee that the order will remain constant over time. This class permits the `null` element.
This class offers constant time performance for the basic operations (`add`, `remove`, `contains` and `size`), assuming the hash function disperses the elements properly among the buckets. Iterating over this set requires time proportional to the sum of the `HashSet` instance's size (the number of elements) plus the "capacity" of the backing `HashMap` instance (the number of buckets). Thus, it's very important not to set the initial capacity too high (or the load factor too low) if iteration performance is important.

public class **TreeSet**
extends AbstractSet
implements SortedSet, Cloneable, Serializable
This class implements the `Set` interface, backed by a `TreeMap` instance. This class guarantees that the sorted set will be in ascending element order, sorted according to the *natural order* of the elements (see `Comparable`), or by the comparator provided at set creation time, depending on which constructor is used.
This implementation provides guaranteed log(n) time cost for the basic operations (`add`, `remove` and `contains`).

public class **TreeMap**
extends AbstractMap
implements SortedMap, Cloneable, Serializable
Red-Black tree based implementation of the `SortedMap` interface. This class guarantees that the map will be in ascending key order, sorted according to the *natural order* for the key's class (see `Comparable`), or by the comparator provided at creation time, depending on which constructor is used.
This implementation provides guaranteed log(n) time cost for the `containsKey`, `get`, `put` and `remove` operations. Algorithms are adaptations of those in Cormen, Leiserson, and Rivest's *Introduction to Algorithms*.

public class **BitSet**
extends Object
implements Cloneable, Serializable
This class implements a vector of bits that grows as needed. Each component of the bit set has a `boolean` value. The bits of a `BitSet` are indexed by nonnegative integers. Individual indexed bits can be examined, set, or cleared. One `BitSet` may be used to modify the contents of another `BitSet` through logical AND, logical inclusive OR, and logical exclusive OR operations.

public abstract class **Dictionary**
extends Object
The `Dictionary` class is the abstract parent of any class, such as `Hashtable`, which maps keys to values. Every key and every value is an object. In any one `Dictionary` object, every key is associated with at most one value. Given a `Dictionary` and a key, the associated element can be looked up. Any non-`null` object can be used as a key and as a value.
NOTE: This class is obsolete. New implementations should implement the Map interface, rather than extending this class.

Properties

```
java.lang.Object
  |
  +----java.util.Dictionary
        |
        +----java.util.Hashtable
              |
              +----java.util.Properties
```

public class Properties extends Hashtable

Used to manage application-specific properties.

The Properties class represents a persistent set of properties. The Properties can be saved to a stream or loaded from a stream. Each key and its corresponding value in the property list is a string. A property list can contain another property list as its "defaults"; this second property list is searched if the property key is not found in the original property list.

public Properties() - Creates an empty property list with no default values.

public Properties(Properties defaults) - Creates an empty property list with the specified defaults.

public String getProperty(String key) - Searches for the property with the specified key in this property list. If the key is not found in this property list, the default property list, and its defaults, recursively, are then checked. The method returns null if the property is not found.

public synchronized void load(InputStream in) throws IOException
Reads a property list from an input stream.

public Enumeration propertyNames() - Returns an enumeration of all the keys in this property list, including the keys in the default property list.

System Properties

Print the current system properties:

```
import java.util.*;

public class SysProps {

      public static void main(String[] args) {
          Properties props = System.getProperties();
          Enumeration e = props.propertyNames();
          while (e.hasMoreElements()) {
                    Object key = e.nextElement();
                    Object value = props.get(key);
                    System.out.println("property " + key + " value " + value);
          }

      }
}
```

The set of system properties always includes values for the following keys:

Key	***Description of Associated Value***
java.version	Java version number
java.vendor	Java vendor-specific string
java.vendor.url	Java vendor URL
java.home	Java installation directory
java.class.version	Java class format version number
java.class.path	Java class path
os.name	Operating system name
os.arch	Operating system architecture
os.version	Operating system version
file.separator	File separator ("/" on UNIX)
path.separator	Path separator (":" on UNIX)
line.separator	Line separator ("\n" on UNIX)
user.name	User's account name
user.home	User's home directory
user.dir	User's current working directory

Reading:

Budd text: Chapter 19

XXII. Quality of an Interface

1. **Cohesion** – all of its methods are related to a single abstraction
e.g.,
```
public class MyContainer {
        public void addItem(Item anItem) {....}
        public Item getCurrentItem() { ... }
        public Item removeCurrentItem() { ... }
        public void processCommand(String command) { ... }
        ...
}
```

Note that *processCommand* might fit better in another class; leave the *MyContainer* class to do what it does best – store *Items*.

2. **Completeness** – interface should support all operations that are a part of the abstraction that the class represents.
e.g.,
The *Stack* class should contain a method to check if the *Stack* is empty

3. **Convenience** – ensure that the interface makes it convenient for clients to do associated tasks, especially common tasks.
e.g.,
Prior to Java 5 *System.in* could not read lines of text or numbers. This was fixed by the addition of the java.util.*Scanner* class

4. **Clarity** – the interface should be clear to programmers
e.g.,
```
LinkedList<String> list = new LinkedList<String>();
list.add("A");
list.add("B");
list.add("C");
...
ListIterator<String> iterator = list.listIterator();
While (iterator.hasNext())
        System.out.println(iterator.next());
...
ListIterator<String> iterator = list.listIterator();  // cursor is just before the first
element
iterator.next();     // advance cursor
iterator.add("X");    // AXBC
...
iterator.remove();     //  the documentation is confusing – is X or A removed?
```

5. **Consistency** – the operations in a class should be consistent with each other with respect to names, parameters and return values, and behavior.
e.g.,
from the Java library:
new GregorianCalendar(year, month-1, day)

the month is between 0 and 11 but the day is between 1 and 31!

XXIII. Java Reflection

Print the notes found at: *http://unixlab.sfsu.edu/~levine/common/csc413/JavaTechTips/*

The link is also posted in *ilearn* under *Course Documents*

XXIV. USING HPROF TO TUNE PERFORMANCE

HPROF - The Java Profiler

- Used to gather runtime information in order to tune performance (re-examine methods that are very time-consuming; perhaps re-code in native code, e.g., C++)
- Included in the Java 2 SDK

C:\java -agentlib:hprof=help

HPROF: Heap and CPU Profiling Agent (JVMTI Demonstration Code)

hprof usage: java -agentlib:hprof=[help]|[<option>=<value>, ...]

Option Name and Value	Description	Default
heap=dump|sites|all	heap profiling	all
cpu=samples|times|old	CPU usage	off
monitor=y|n	monitor contention	n
format=a|b	text(txt) or binary output	a
file=<file>	write data to file	java.hprof[.txt]
net=<host>:<port>	send data over a socket	off
depth=<size>	stack trace depth	4
interval=<ms>	sample interval in ms	10
cutoff=<value>	output cutoff point	0.0001
lineno=y|n	line number in traces?	y
thread=y|n	thread in traces?	n
doe=y|n	dump on exit?	y
msa=y|n	Solaris micro state accounting	n
force=y|n	force output to <file>	y
verbose=y|n	print messages about dumps	y

Obsolete Options
---
gc_okay=y|n

Examples

- Get sample cpu information every 20 millisec, with a stack depth of 3:
  java -agentlib:hprof=cpu=samples,interval=20,depth=3
  com.develop.demos.TestHprof
- Get heap usage information based on the allocation sites:
  java -agentlib:hprof=heap=sites com.develop.demos.TestHprof

Notes

 - The option format=b cannot be used with monitor=y.
 - The option format=b cannot be used with cpu=old|times.
Warnings

 - This is demonstration code for the JVMTI interface and use of BCI,
   it is not an official product or formal part of the J2SE.
 - The option format=b is considered experimental, this format may change
   in a future release.

C:\Courses\413\PerfAnal>java -agentlib:hprof=cpu=samples,depth=6
com.develop.demos.TestHprof
Total run time of 1672 milliseconds
Dumping CPU usage by sampling running threads ... done.

Notes:
cpu=samples
The VM regularly pauses execution to record the call stack trace (shows which method is executing, etc.)
depth=6
Only record up to the top 6 levels of the stack in the trace

Following are some sample traces. Note that each stack trace is identified with a number.
TRACE 89:
java/lang/StringBuffer.<init>(StringBuffer.java:120) ***<== init indicates a constructor call***
java/lang/StringBuffer.<init>(StringBuffer.java:134)
com/develop/demos/TestHprof.addToCat(TestHprof.java:17)
com/develop/demos/TestHprof.makeString(TestHprof.java:12)
com/develop/demos/TestHprof.main(TestHprof.java:54)
TRACE 151:
java/lang/StringBuffer.<init>(StringBuffer.java:120)
java/lang/StringBuffer.<init>(StringBuffer.java:134)
com/develop/demos/TestHprof.makeStringWithLocal(TestHprof.java:30)
com/develop/demos/TestHprof.main(TestHprof.java:56)
TRACE 153:
java/lang/System.arraycopy(System.java:Native method)
java/lang/String.getChars(String.java:549)
java/lang/StringBuffer.append(StringBuffer.java:397)
java/lang/StringBuffer.<init>(StringBuffer.java:135)
com/develop/demos/TestHprof.makeStringWithLocal(TestHprof.java:30)
com/develop/demos/TestHprof.main(TestHprof.java:56)

We will use the following program for demonstration purposes. We are interested in optimizing String concatenations.

```
package com.develop.demos;

import java.io.IOException;

public class TestHprof {
   public static String cat = null;
   public final static int loop=5000;

   public static void makeString() {
      cat = new String();
      for (int n=0; n<loop; n++) {
         addToCat("more");
      }
   }

   public static void addToCat(String more) {
      cat = cat + more;
   }

   public static void makeStringInline() {
      cat = new String();
      for (int n=0; n<loop; n++) {
         cat = cat + "more";
      }
   }

   public static void makeStringWithLocal() {
      String tmp = new String();
      for (int n=0; n<loop; n++) {
        tmp = tmp + "more";
      }
      cat = tmp;
   }

   public static void makeStringWithBuffer() {
      StringBuffer sb = new StringBuffer();
      for (int n=0; n<loop; n++) {
         sb.append("more");
      }
      cat = sb.toString();
   }
   public static void main(String[] args) {
         long begin = System.currentTimeMillis();
```

```
        if (null != System.getProperty("WaitForProfiler")) {
                System.out.println("Start your profiler, then press any key to begin...");
                try {
                        System.in.read();
                }
                catch (IOException ioe) {
                }
        }

        makeString();
        makeStringInline();
        makeStringWithLocal();
        makeStringWithBuffer();

        long end = System.currentTimeMillis();
        System.out.println("Total run time of " + (end - begin) + " milliseconds");
    }
}
```

Program Notes:
We are interested in the performance of the four methods:

```
makeString();
makeStringInline();
makeStringWithLocal();
makeStringWithBuffer();
```

Each method builds a String wherein "more" is concatenated 5000 times.

1. *makeString()* - calls method *addToCat* for each concatenation using the + operator to concatenate to a global variable.
2. *makeStringInline()* - attempt to improve concatenations by avoiding the call to *addToCat* (concatenations are done within the method)
3. *makeStringWithLocal()* - attempt to improve the operation of *makeStringInline()* by using a local variable for all concatenations
4. *makeStringWithBuffer()* - use a StringBuffer and appends to avoid concatenations

First, *TestHprof* is compiled. Then *TestHprof* is executed with profiling turned on:

```
C:\Courses\413\PerfAnal>java -agentlib:hprof=cpu=samples,depth=6
                                                com.develop.demos.TestHprof
```

Note that the default logging file is used: ***java.hprof.txt***

The following summary information was provided at the bottom of the log file.

```
CPU SAMPLES BEGIN (total = 5064)
```

rank	self	accum	count	trace	method
1	16.86%	16.86%	854	89	java/lang/StringBuffer.<init>
2	16.84%	33.71%	853	151	java/lang/StringBuffer.<init>
3	16.84%	50.55%	853	122	java/lang/StringBuffer.<init>
4	11.85%	62.40%	600	153	java/lang/System.arraycopy
5	11.79%	74.19%	597	96	java/lang/System.arraycopy
6	11.73%	85.92%	594	130	java/lang/System.arraycopy
7	0.47%	86.39%	24	86	java/lang/StringBuffer.<init>
8	0.39%	86.79%	20	119	java/lang/StringBuffer.append

The traces are ranked in order of the time they are active (*self*). The active method for the trace is indicated. For example, *java/lang/StringBuffer.<init>* indicates that 16.86% of the time sampling was done the *StringBuffer* constructor was active - the stack trace was given in trace 89.

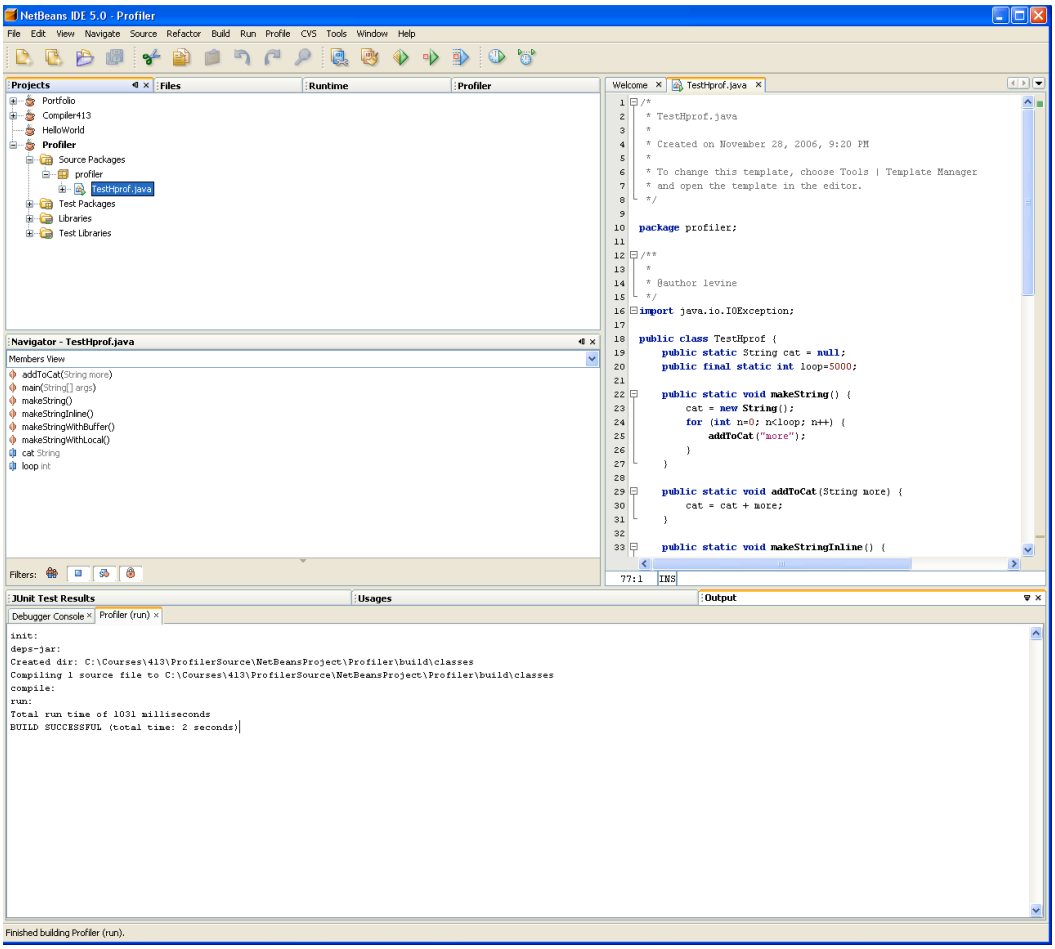
TRACE 89:

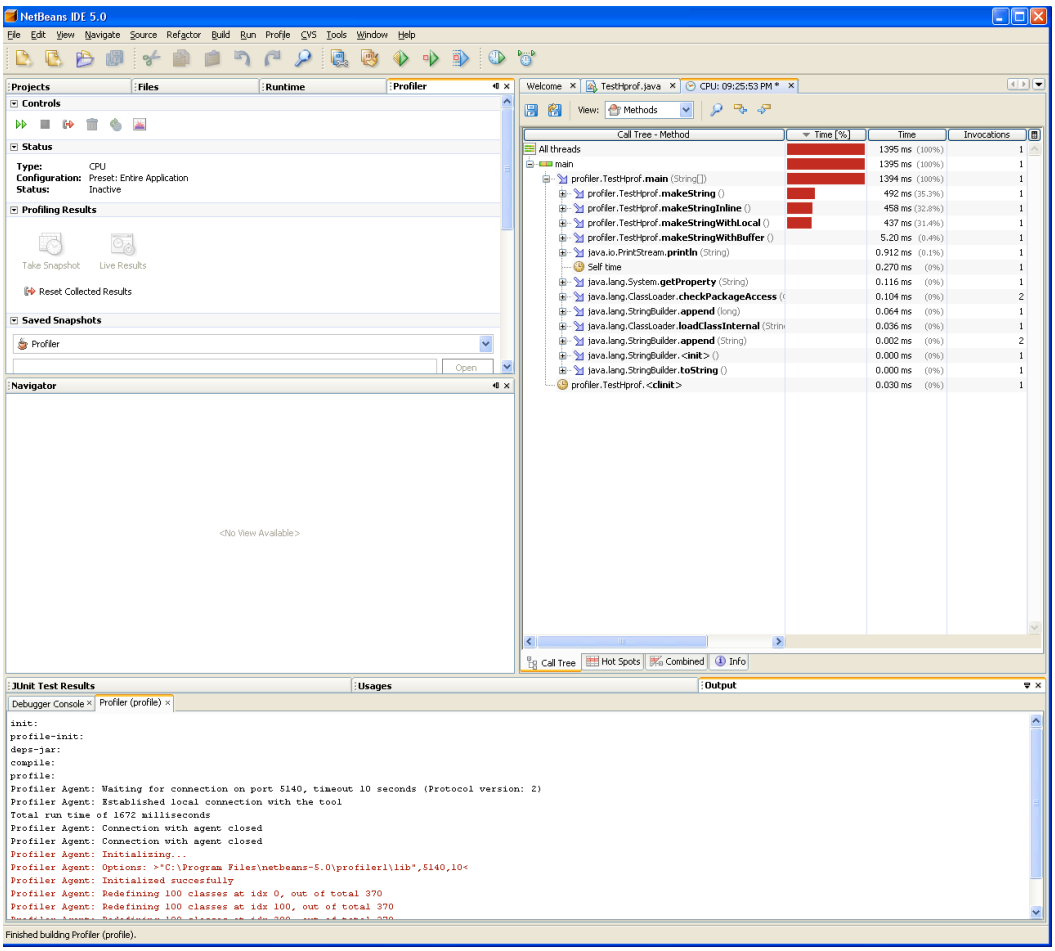
java/lang/StringBuffer.<init>(StringBuffer.java:120)
java/lang/StringBuffer.<init>(StringBuffer.java:134)
com/develop/demos/TestHprof.addToCat(TestHprof.java:17)
com/develop/demos/TestHprof.makeString(TestHprof.java:12)
com/develop/demos/TestHprof.main(TestHprof.java:54)

This trace indicates that *makeString* was active - the 16.86% timing belongs to *makeString*. We also find that *makeString* was active in trace 96. These two traces account for approximately 30% of the CPU time.
Similar analyses indicate that *makeStringInline* and *makeStringWithLocal* each account for approximately 30% of the time.

Finally, we note that .7% of the CPU time is consumed by *makeStringWithBuffer*. Based on these analyses we conclude that we should use *StringBuffers* rather than normal String concatenation when concatenating Strings repeatedly.

NetBeans GUI




NetBeans IDE 5.0
File Edit View Navigate Source Refactor Build Run Profile CVS Tools Window Help
Projects
Files
Runtime
Profiler
Controls
Status
Type: CPU
Configuration: Preset: Entire Application
Status: Inactive
Profiling Results
Take Snapshot
Live Results
Reset Collected Results
Saved Snapshots
Profiler
Open
Navigator
<No View Available>
Welcome
TestHprof.java
CPU: 09:25:53 PM *
View: Methods
Call Tree - Method
Time (%)
Time
Invocations
All threads 1395 ms (100%) 1
main 1395 ms (100%) 1
profiler.TestHprof.main (String[]) 1394 ms (100%) 1
profiler.TestHprof.makeString () 492 ms (35.3%) 1
profiler.TestHprof.makeStringInline () 458 ms (32.8%) 1
profiler.TestHprof.makeStringWithLocal () 437 ms (31.4%) 1
profiler.TestHprof.makeStringWithBuffer () 5.20 ms (0.4%) 1
java.io.PrintStream.println (String) 0.912 ms (0.1%) 1
Self time 0.270 ms (0%) 1
java.lang.System.getProperty (String) 0.116 ms (0%) 1
java.lang.ClassLoader.checkPackageAccess 0.104 ms (0%) 2
java.lang.StringBuilder.append (long) 0.064 ms (0%) 1
java.lang.ClassLoader.loadClassInternal 0.036 ms (0%) 1
java.lang.StringBuilder.append (String) 0.002 ms (0%) 2
java.lang.StringBuilder.<init> () 0.000 ms (0%) 1
java.lang.StringBuilder.toString () 0.000 ms (0%) 1
profiler.TestHprof.<clinit> 0.030 ms (0%) 1
Call Tree
Hot Spots
Combined
Info
JUnit Test Results
Usages
Output
Debugger Console
Profiler (profile)
init:
profile-init:
deps-jar:
compile:
profile:
Profiler Agent: Waiting for connection on port 5140, timeout 10 seconds (Protocol version: 2)
Profiler Agent: Established local connection with the tool
Total run time of 1672 milliseconds
Profiler Agent: Connection with agent closed
Profiler Agent: Connection with agent closed
Profiler Agent: Initializing...
Profiler Agent: Options: >"C:\Program Files\netbeans-5.0\profiler1\lib",5140,10<
Profiler Agent: Initialized succesfully
Profiler Agent: Redefining 100 classes at idx 0, out of total 370
Profiler Agent: Redefining 100 classes at idx 100, out of total 370
Finished building Profiler (profile).

Appendix A: Lab Submissions - jar File and Zipping

Creating a jar file in Netbeans:
Assume you are creating a project in C*:\Courses\413\NetBeansProjects* with project name of *FirstProject*

1. Develop your application until it is working and ready to submit
2. Click on the ***Clean and Build Project*** icon on the task bar
3. This will create a jar file in the dist subdirectory of your project (*C:\Courses\413\NetBeansProjects\FirstProject\dist*)

Now, you can open a command window and execute the jar file as:
java -jar <jar-file-name> <command line args>
e.g.
java -jar FirstProject.jar john mary

If you are required to submit a zipped file for your assignment (**required for all assignments, except where noted**) then you should create the following zip file:

1. Create a directory with the following name: <lastName, firstName-section> e.g., create a directory SmithJohn-1
2. Include your jar file in the directory of step 1. The jar file name will be created from the assignment name
3. Include all of your source files in subdirectory(ies) off the directory of step 1 (include the Netbeans src directory for your project).
4. Include the test file, when provided, with the name "test.txt"
5. Zip the directory from step 1 and give it the same name as the directory

e.g.,
If your name is John Smith, you're in section 2 and you are submitting the lexer lab then create the following structure:

```
SmithJohn-2/
   lexer.jar
   src/
      lexer/
         Lexer.java
         SourceReader.java
         Symbol.java
         Token.jave
         Tokens.java
         TokenType.jave
```

All of these files should be zipped into a file named SmithJohn-2.zip.

Be sure to check your jar file works correctly prior to submission by executing it in a command window. If you submit your deliverables with a poorly formed jar file then you will receive no credit for the assignment.

Appendix B: Computer Science 413 Exercises

```
YOU MUST USE APPROPRIATE INDENTATION (SUCH AS THAT PRODUCED BY
NETBEANS). Do not use tabs that will spread source code over several
lines, which will make it hard to read:

public void static main(String args[]) {
                 if (SourceReader.get(i)) {
                                        System.out.println(
          "this is not good");
                    }
}

Note that the following is much easier to read:

public void static main(String args[]) {
     if (SourceReader.get(i)) {
         System.out.println("this is not good");
     }
}

Please DO NOT provide source code which is very difficult to read. I
cannot ask the grader to attempt to read a program with poor
indentation, as illustrated above. Please be very careful about this
situation.

Be sure to remove ALL debug printout, except when specifically
requested.
```

1.(5 points)
A. Consider the following Java program:

```
import java.lang.*;
public class SecondProgram {
    public static void main ( String args[] ) {
          if (args.length > 0)  {
               System.out.println("Hello " + args[0]);
           } else {
               System.out.println( "Hello everybody!");
           }
      }
}
```

Modify the program such that if there are no command line arguments then it will output:
Hello everybody!

e.g.
java SecondProgram
Hello everybody!

If there are command line arguments (names) then it will print Hello to all the names with one name printed per line (along with the number of the name):

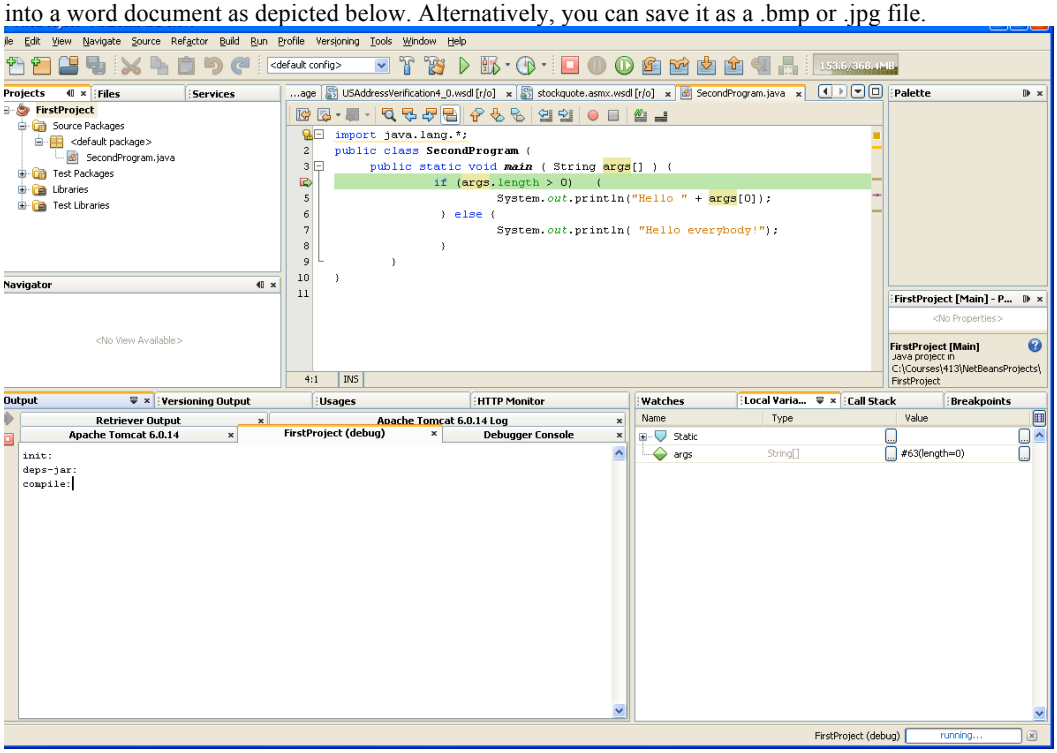
java SecondProgram Helen Armen Joseph

Hello
1. Helen
2. Armen
3. Joseph

java SecondProgram Helen
Hello
1. Helen

B. I would like everyone to run the netbeans debugger on the program in Part A.

Set a breakpoint at the *if* statement and then execute in the debug mode. WHEN THE PROGRAM STOPS AT THE LINE WITH THE BREAKPOINT take a screen snapshot (alt-printscreen) and paste it into a word document as depicted below. Alternatively, you can save it as a .bmp or .jpg file.



Create a directory as described above, e.g., SmithJohn-2/, which should only include ***two files***. The first file should include your .java code. The second file will contain the screen snapshot file. Submit the zip file of this directory to ilearn.

2. (15 points) Program the ***Towers of Hanoi*** problem in Java:

Given:

- n disks, all of different sizes - the size of the i^{th} disk is i
- 3 pegs - A, B, C the number of pegs might change in a future version of the game
- Initially, all the disks are stacked on peg A

Requirement:
Never place a larger disk on top of a smaller disk (disk 5 is larger than disk 4, etc.)

Objective:
Move the disks from peg A to peg C

Input:
n, the number of disks initially stacked on peg A; you should get n from the command line, as a command line argument; assume n < 10

Output:
Commands describing how the player should move the pegs

Example:

If 3 is input then the output should be:

Move	*Peg Configuration*		
	A	*B*	*C*
init	*3 2 1*		
1 from A to C	*3 2*		*1*
2 from A to B	*3*	*2*	*1*
1 from C to B	*3*	*2 1*	
3 from A to C		*2 1*	*3*
1 from B to A	*1*	*2*	*3*
2 from B to C	*1*		*3 2*
1 from A to C			*3 2 1*

Strategy:
If there are n disks on peg A and we need to move them to peg C then we'll use peg B as a utility peg:
Move the top n-1 disks from A to B
Move the n^{th} disk from A to C
Move the n-1 disks from B to C, using peg A as a utility
OR
If we must move n disks on the *source* peg to the *dest* peg using a *utility* peg:
Move the top n-1 disks from *source* to *utility*
Move the n^{th} disk from *source* to *dest*
Move the n-1 disks from *utility* to *dest*, using peg *source* as a *utility*

Be sure to choose appropriate classes - it should require minimal changes if we desire to modify the problem to include any number of pegs (you should provide a flexible, extensible solution) or modify our strategy. Note your class cohesion and degree of coupling between classes (maximize cohesion, minimize coupling)

You are expected to think about and correctly identify the objects in this exercise. The reasons I assigned this exercise are for you:
- To get more practice programming in Java
- To think about and utilize Object Oriented Programming techniques

The second reason requires you to "think outside the box"...you must be very careful to first identify the objects in the problem space (don't discount identifying an object because you might think it's too trivial), form their associated classes, discover their associated properties (instance variables) and roles (methods).
Herein, it's extremely important to assign roles to objects based on their expertise - e.g. if you want me to drive to school then you should not do the job for me, you should ask ME to drive and I'll determine HOW to satisfy the request.
All classes should be included in a singe file.

You are not asked to solve the problem for n pegs.....just be sure that if, in the future, you are asked to extend the problem to n pegs it will not require major code modifications..this requires careful thinking about the objects involved in the computation and their classes

3. (15 points) Write a Java program with a test harness class *EvaluatorTester* which calls on *Evaluator* for evaluating simple expressions. You can use the following definition of the test harness (note the command line arguments are a series of expressions, separated by spaces, to be evaluated by *Evaluator*):

```
public class EvaluatorTester {

   public static void main(String[] args) {
      Evaluator anEvaluator = new Evaluator();
      for (String arg : args) {
         System.out.println(arg + " = " + anEvaluator.eval(arg));
      }
   }
```

I have provided an *almost correct* version of the *Evaluator* class. You should program the utility classes it uses - *Operand* and *Operator* - and then follow my suggestion on how to correct the *Evaluator* class. Note that the expressions are composed of integer operands and operators drawn from +, -, * and / Following is the *almost correct* version of *Evaluator* source code:

```
import java.util.*;

public class Evaluator {

   private Stack<Operand> opdStack;
   private Stack<Operator> oprStack;

   public Evaluator() {
      opdStack = new Stack<Operand>();
      oprStack = new Stack<Operator>();
   }

   public int eval(String expr) {
      String tok;

            // init stack - necessary with operator priority schema;
            // the priority of any operator in the operator stack other then
            // the usual operators - "+-*/" - should be less than the priority
            // of the usual operators
      oprStack.push(new Operator("#"));

                  delimiters = "+-*/#! ";

      StringTokenizer st = new StringTokenizer(expr,delimiters,true);
            // the 3rd arg is true to indicate to use the delimiters as tokens, too
            // but we'll filter out spaces

      while (st.hasMoreTokens()) {
         if ( !(tok = st.nextToken()).equals(" ")) {          // filter out spaces
            if (Operand.check(tok)) {                          // check if tok is an operand
               opdStack.push(new Operand(tok));
            } else {
               if (!Operator.check(tok)) {
```

```
                System.out.println("*****invalid token******");
                System.exit(1);
            }

            Operator newOpr = new Operator(tok);        // POINT 1

            while ( ((Operator)oprStack.peek()).priority() >= newOpr.priority()) {

                        // note that when we eval the expression 1 - 2 we will
                        // push the 1 then the 2 and then do the subtraction operation
                        // This means that the first number to be popped is the
                        // second operand, not the first operand - see the following code
                Operator oldOpr = ((Operator)oprStack.pop());
                Operand op2 = (Operand)opdStack.pop();
                Operand op1 = (Operand)opdStack.pop();
                opdStack.push(oldOpr.execute(op1,op2));
            }

            oprStack.push(newOpr);
          }
        }
          // Control gets here when we've picked up all of the tokens; you must add
          // code to complete the evaluation - consider how the code given here
          // will evaluate the expression    1+2*3
          // When we have no more tokens to scan, the operand stack will contain 1 2
          // and the operator stack will have + *   with 2 and * on the top;
          // In order to complete the evaluation we must empty the stacks (except
          // the init operator on the operator stack); that is, we should keep
          // evaluating the operator stack until it only contains the init operator;
          // Suggestion: create a method that takes an operator as argument and
          // then executes the while loop; also, move the stacks out of the main
          // method
      }
    }
}
```

The algorithm processes operand and operator tokens. If an operand token is scanned then it is immediately pushed on the operand stack. If a new operator token is scanned then its priority is compared to the priority of the operator on top of the operator stack. As long as the priority of the new operator token is smaller than the priority of the operator on the stack the following occurs:
1. The stack operator is popped
2. It's operands are popped from the operand stack
3. The operation is performed and the result is pushed onto the operand stack

We use two bogus operators *new Operator("#")* and *new Operator("!")* to avoid numerous checks for empty operator stack. I have described above how to use "#" to assist in the algorithm. Consider how to use "!" as an operator placed at the end of the input stream to facilitate the logic (recall, the use of "!" was described in lecture.Use the following operator priorities:

Operator	**Priority**
#	0
!	1
+ -	2
* /	3

```
StringTokenizer st = new StringTokenizer("these are several tokens");
while (st.hasMoreTokens()) {
   System.out.println(st.nextToken());
}
```

prints the following output:

these
are
several
tokens

Notes:

- *hasMoreTokens()* is used to ensure that before we try to get the next token (*String*) we actually have one available.
- *nextToken()* is used to obtain the next token (*String*); note we should always check that there are more items remaining prior to calling on *nextToken()* else we can have problems
- You should only use 2 files, one file for the test harness (*EvaluatorTester*) and the other file to contain **all other classes** (*Evaluator*, etc.).

Hints:
Examine the Java documentation on the *String* and *Stack* classes.
Use the Netbeans debugger - it will provide a lot of help in this and other problems!

Include the following methods for the indicated classes:

Operator

abstract int priority()
boolean check(String tok)
abstract Operand execute(Operand opd1, Operand opd2)

Operand

Operand(String tok)
Operand(int value)
boolean check(String tok)
int getValue()

Note: Consider the possibility of adding new operators. How would you design your program to easily extend the set of operators, and, ***without using a switch*** (using switch statements is not a good policy). Make operator an *abstract superclass*; **use one class per operator**; init a HashMap with all of the operators and their instances. ***You are required to create/use this operator hierarchy for this exercise***.

Don't worry about poorly formed expressions - that is, no need to check for those cases (assume the input is correct).
You may assume that we are using integer division – truncate the result.

Following is some pseudocode that provides some ideas on solving the problem.
Operator should be an abstract class, with method declarations and members like these:

```
abstract class Operator {
   public abstract Operand execute();
   .....
}
```

This class is then extended to the various operators-- Addition, Subtraction etc.

```
class AdditionOperator extends Operator{
   public Operand execute(Operand op1, Operand op2){
   //do an addition, return the sum
```

```
    }
    ...
}
```

The Operator class should contain an instance of a HashMap. This map will use as keys the tokens we're interested in, and values will be instances of the Operators. ex:

```
HashMap operators = new HashMap();

operators.put("+",new AdditionOperator());

operators.put("-",new SubtractionOperator());

etc....
```

Next, the Operator class will make use of the HashMap. Whenever we need a new operator we will access the HashMap. Because the Operator class is abstract, it can't be instantiated. So instead of

```
Operator newOpr= new Operator(tok);

you might do this:

Operator newOpr = (Operator) operators.get(tok);
```

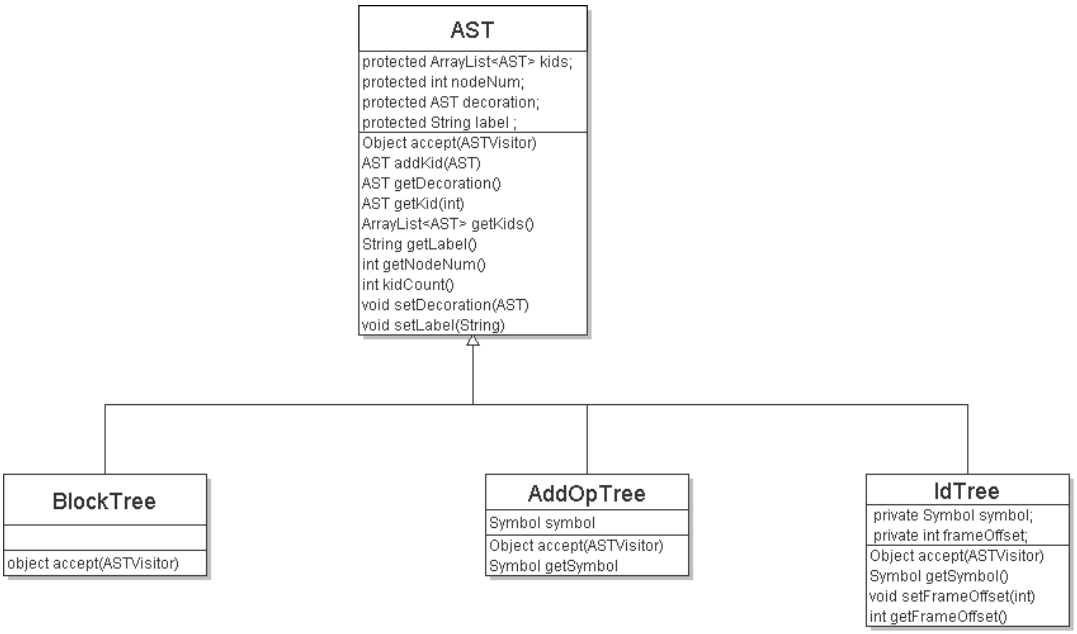
newOpr now "knows" what type of operator it is, and can act accordingly when the *execute()* method (or other relevant methods) is called on it.

ex:

```
String tok="+";

Operator newOpr = (Operator) operators.get(tok);  //newOpr is an
AdditionOperator

opdStack.push(newOpr.execute(op1,op2)); //addition taking place
```

Creating a Class Diagram for AST class hierarchy

(following is a partial class hierarchy for the AST classes)



You can download the UML plugin for Netbeans – the plugin will provide much help in developing class diagrams.

Compiler Exercises (always provide Javadoc comments in your code):
The rules given in bold in the following grammar have been added to the course reader's version of the compiler.

1. Modified Grammar for X

```
PROGRAM -> ‘program’ BLOCK                ==> program

BLOCK -> ‘{‘ D* S* ‘}’                    ==> block

D -> TYPE NAME                            ==> decl
   -> TYPE NAME FUNHEAD BLOCK             ==> functionDecl

TYPE  ->  ‘int’
        ->  ‘boolean’
        -> ‘float’
        -> ‘void’

FUNHEAD  -> '(' (D list ',')? ')'         ==> formals

S -> ‘if’ E ‘then’ BLOCK ‘else’ BLOCK     ==> if
  -> ‘whil’e E BLOCK                      ==> while
  -> ‘return’ E                           ==> return
  -> ‘return’;                            ==> return
  -> BLOCK
  -> NAME  ’=’  E                         ==> assign
  -> NAME ‘(‘ (E list ‘,’)? ‘)’           ==> call

E -> SE
  -> SE ‘==’ SE                           ==> =
  -> SE ‘!=’ SE                           ==> !=
  -> SE ‘<’  SE                           ==> <
  -> SE ‘<=’ SE                           ==> <=
  -> SE ‘>’ SE                            ==> >
  -> SE ‘>=’ SE                           ==> >=

SE  ->  T
     ->  SE ‘+’ T                         ==> +
     ->  SE ‘-‘ T                         ==> -
     ->  SE ‘|’ T                         ==> or

T  -> F
   -> T ‘*’ F                             ==> *
   -> T ‘/’ F                             ==> /
   -> T ‘&’ F                             ==> and

F  -> ‘(‘ E ’)’
   -> NAME
   -> <int>
   -> <float>
   -> NAME '(' (E list ',')? ')'          ==> call
   -> ‘!’F                                ==> not

NAME  -> <id>
```

2. Modify the Lexer to:

(15 points for segments of the following)

A. Recognize the new tokens: > >= ! *void float* <*float*> (floating literals conform to: d+.d* or d*.d+ that is,
either 1 or more digits followed by a dot and optionally followed by 0 or more digits or
0 or more digits followed by a dot followed by 1 or more digits). Check the FAQs for more on scanning float literals.

Be sure to change *token file* appropriately and then run TokenSetup as required
Also, be sure to provide a command line argument that is the file name which contains the input stream (not simple.x).

B. Change comments from *// to end-of-line* to - - *to end-of-line slashes* to end-of-line modified to *dashes* to end-of-line

C. Include line number information within tokens for subsequent error reporting, etc.

D. Change id's from those conforming to Java to: [A-Za-z][A-Za-z0-9]*

E. Output the source program with line numbers (since SourceReader reads the source lines it should save the program in memory for printout after the tokens are scanned):
e.g.

1. program { int i int j
2. i = 2
3. j = 3
4. i = write(j+4)
5. }

F. Print each token with line number

```
READLINE: program { int i int j
program   left: 0 right: 6 line: 1
{         left: 8 right: 8: line: 1
int       left: 10 right: 12 line: 1
i         left: 14 right: 14 line: 1
int       left: 16 right: 18 line: 1
j         left: 20 right: 20 line: 1
.
.
```

G. Modify the Exception processing appropriately - e.g. if we check for *Exception* when we can only get an *IOException* then change the *catch* clause from *catch (Exception...* to *catch (IOException...*

The main method in Lexer.java uses a specific file (e.g. t or simple.x) for processing tokens...modify the code to use a command line argument for the file for processing tokens..

Comments on this lab..

1. TokenSetup should NOT be changed
2. The tokens file should be changed appropriately
3. Use the main method in Lexer.java for testing - DO NOT use other packages in the compiler (e.g. Compiler.java) since there will always be complaints due to not recognizing the new constructs
4. I will post your test case just prior to the due.
5. REMOVE ALL DEBUG statements not required

If you encounter an error, e.g. you find a "." on line 7 of the source file that contains 20 lines, then you should:

1. report the error
2. stop processing tokens at that point
3. echo the lines of the source file with line numbers up to and including the error line - e.g., echo lines 1 through 7 inclusive in the case with the "." on line 7
4. exit

3. Modify the Parser to:

(15 points for segments of the following)

A. Parse the new rules specified in the modified grammar and build the indicated AST's

B. Provide better diagnostics when detecting a syntax error - e.g. provide the source line number, pointer to an offending token (if the expected token wasn't found then point to the offending token and indicate the token that was expected)

4. Build a Parser for the Following Grammar:

CLASSDEF	->	MODIFIER ‘class’ NAME ‘{‘ CLASSDECL* ‘}’	==> classdef
MODIFIER	->	‘public’	
	->	‘private’	
CLASSDECL	->	D	==> classdecl
	->	METHOD	==> classdecl
D	->	TYPE NAME	==> decl
TYPE	->	‘int’	
	->	‘boolean’	
METHOD	->	METHODPROFILE ‘{‘ METHODBODY ‘}’	==> method
METHODPROFILE	->	MODIFIER TYPE NAME METHODHEAD	==> profile
	->	MODIFIER NAME METHODHEAD	==> profile
METHODHEAD	->	'(' (D list ',')? ')'	==> formals
METHODBODY	->	BODYITEM*	==> body
BODYITEM	->	D	
	->	S	
S	->	‘if’ E ‘then’ BLOCK ‘else’ BLOCK	==> if
	->	‘while’ E BLOCK	==> while
	->	‘return’ E	==> return
	->	NAME ‘=’ E	==> assign
BLOCK	->	‘{‘ D* S* ‘}’	==> block
E	->	SE	
	->	SE ‘==’ SE	==> =
	->	SE ‘!=’ SE	==> !=
SE	->	T	
	->	SE ‘+’ T	==> +
	->	SE ‘-‘ T	==> -
T	->	F	
	->	T ‘*’ F	==> *
F	->	‘(‘ E ‘)’	
	->	NAME	
	->	<int>	
	->	NAME '(' (E list ',')? ')'	==> call
NAME	->	<id>	

Notes:

- Embedded class definitions are not allowed
- Embedded method definitions are not allowed
- Constructor methods are distinguished from other methods - a return type is not specified and the names of the constructor and class are the same

5. Modify the Constrainer to:

A. Constrain the new rules specified in the modified grammar and decorate the indicated AST's
   Note that function return types can be void - in this case the function isn't returning any values
B. Check that all functions have at least one return statement
C. Add the *true* and *false* literal values

6. Modify the Code Generator to:

A. Generate code for the new rules specified in the modified grammar and decorate the indicated AST's; you will need new bytecodes for the new operators

B. Evaluate all constant expressions at compile time -
   e.g. replace *while (1 == 1) ...* with *while (true)*

C. Initialize variables to -1 rather than 0

D. Initialize the frame variable at offset *i* in the frame to i

```
program { int i int j                 init i to 0, j to 1
  int f(int i) { int j int k          init j to 1, k to 2 (i will have the arg value of 3)
      return i + j + k + 2
  }
  int m                               init m to 2
  m = f(3)
  i = write(j + m)
}
```

7. Modify the Compiler to Use applets to allow users to compile and execute X programs over the web

8. Consider the following sub-grammar of the X-compiler:

```
D -> TYPE NAME                          ==> decl
   -> TYPE NAME FUNHEAD BLOCK           ==> functionDecl

TYPE  ->  'int'
        ->  'boolean'

FUNHEAD  -> '(' (D list ',')? ')'       ==> formals
```

Note that the formals is a *list of declarations*, including both variable and function declarations. Unfortunately, we *do not want to include a function declaration* as a formal parameter. Among the solutions to this problem are:

A. Check for any formal parameter function declarations during ***constraining*** and generate an error condition if one is found (ok, not a great solution) or

B. ***Modify the grammar*** (which is really an *important documentation tool* for the language) so these formal parameter function declarations cannot occur (a very good solution). *It would be best not to invent new AST's* in this solution. The following sub-grammar provides a solution that satisfies this criteria:

```
D -> VARDECL
   -> TYPE NAME FUNHEAD BLOCK           ==> functionDecl

VARDECL -> TYPE NAME                    ==> decl

TYPE  ->  'int'
        ->  'boolean'

FUNHEAD  -> '(' (VARDECL list ',')? ')'         ==> formals
```

Note that this solution still allows the use of *embedded function declarations*; that is, BLOCK's can still contain function delcarations. The added difficulty with this solution is found in the new Parser code - the code for the *declaration function* must initially call the ***VARDECL*** function and then *deconstruct the returned decl tree* if a function declaration is being parsed (not too difficult).

Solve the function declaration problem using either technique described above.