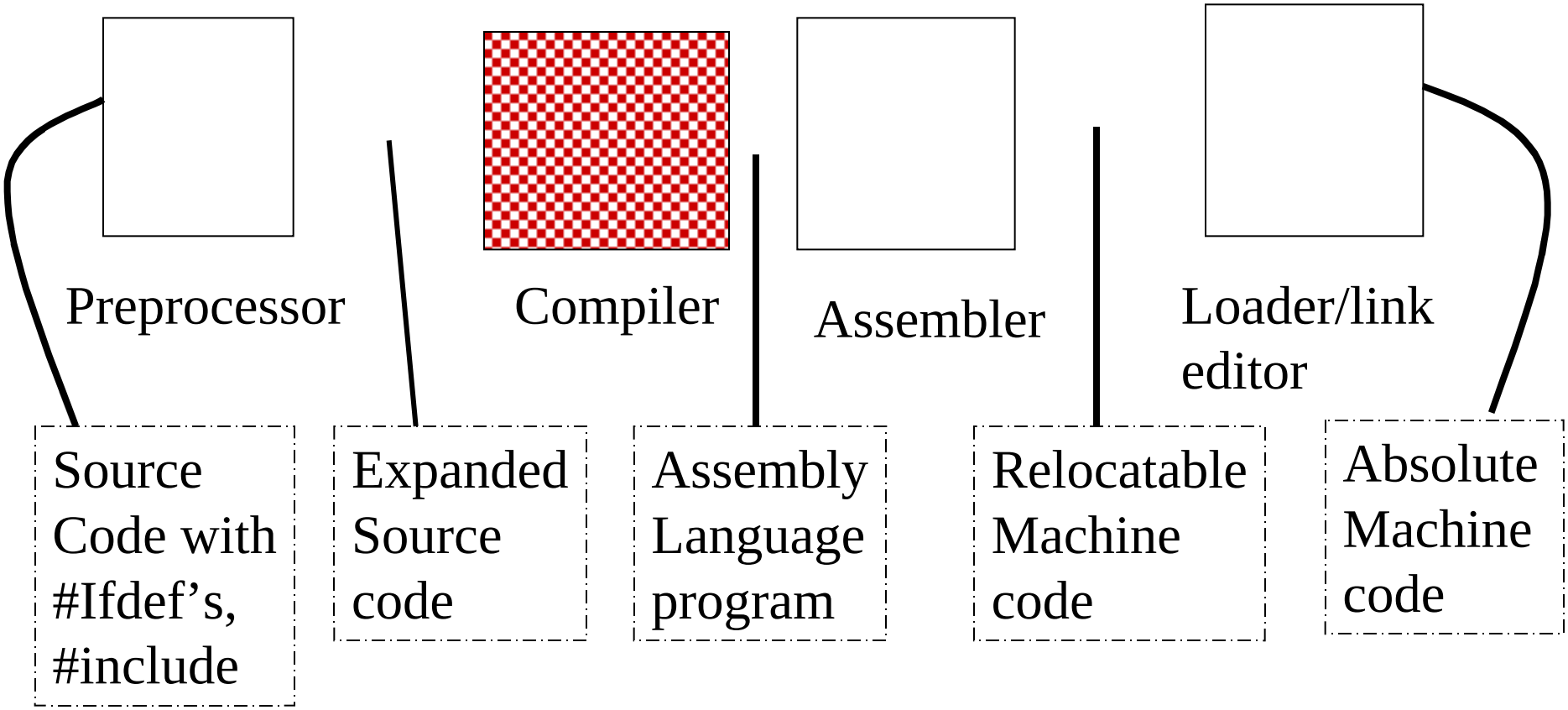
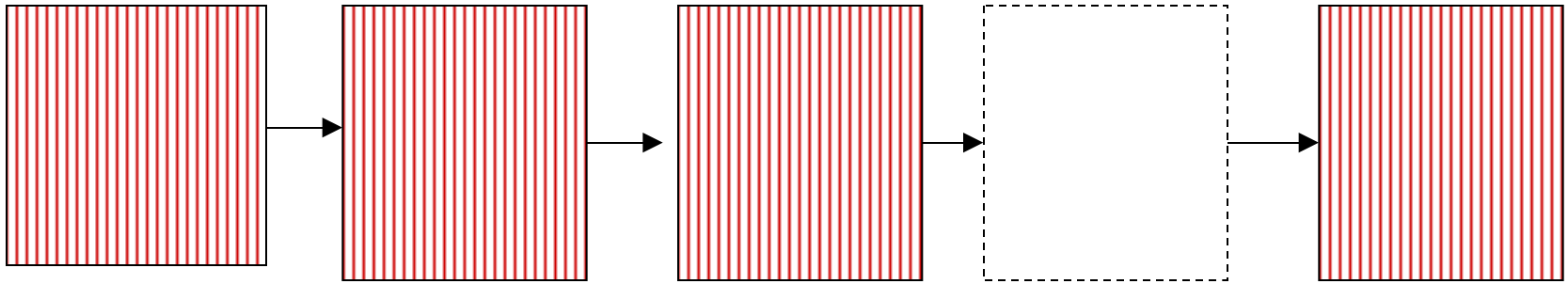
Lexical Analysis-1

- **Compilers**
- **Our first project**
- **Tokens**
- **Regular expressions**

Compilers and their context



Inside a Compiler



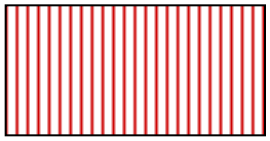
Lexical Analyzer (scanner)

Parser (Syntax Analyzer)

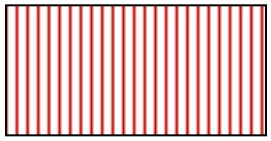
Semantic Analyzer

Optimizer (optional)

Code Generator



Error handler



Symbol Table Manager

All above passes communicate with this layer.

Course Overview

- **Learn by doing - project orientation; supplemented by theoretical underpinings - weekly problem assignments**
- **Some familiarity with object-oriented programming essential (C++, Java, ST80)**
- **Prerequisites are essential**
- **Class webpage and newsgroup**
  http://www.cs.rutgers.edu/~ryder/415
  ru.nb.dcs.class.415

First Project: Side-Effect-free Interpreter

- **Getting familiar with Java**
  - **javac: Java → Java byte code**
  - **java: Java interpreter**
  - **jdb: Java debugger (Must compile .java files with javac -g)**
- **Class directory on remus:**
  /usr/local/class/cs415/sp99/
- **Intermediate compiler representations are often trees, so this assignment practices tree walking in Java**

Grammar Exerpt

Appel, Ch 1

Stm ::= Stm ; Stm %%compound statement

Stm ::= *id* := Exp %%assignment statement

Stm ::= *print* (ExpList) %%print statement

Exp ::= *num* %%NumExp

Exp ::= Exp Binop Exp %%OpExp

Binop ::= + %%Plus | - %%Minus | *%%Times | / %%Div

Java Classes

```
public abstract class Stm{}
public class CompoundStm extends Stm
{  public Stm stm1, stm2;
   public CompoundStm(Stm s1, Stm s2)
   { stm1=s1; stm2=s2;}}
```

Stm ::= Stm ; Stm

```
public class AssignStm extends Stm
{  public String id; public Exp exp;
   public AssignStm(String i, Exp e)
   {   id = i; exp = e;}  }
```

Stm ::= *id* := Exp

```
public abstract class Exp{}
```

```
public class NumExp extends Exp
{  public int num;
   public NumExp(int n) {num = n;}  }
```

Exp ::= *num* %%NumExp

First Project

- **Given a simple programming language (PL) grammar with binary expressions, prints and sequences of statements**
- **Identify each nonterminal with an abstract class**
- **Extend abstract class by 1 subclass per production; can think of each instance variable of the subclass as a tree root if it's a nonterminal or as a leaf if it's a terminal**

First Project Conventions

1. Trees are described by a grammar
2. Each nonterminal in grammar corresponds to an abstract class
3. Each production has 1 corresponding class
4. For each nontrivial symbol on rhs of production, there is a field in this class
5. Every class has a constructor for initializing fields
6. Data fields are immutable.

First Project

- **To traverse the tree you will need to identify the type of each node as you encounter it**
- **To interpret the program, you will define a *Table* object, essentially a list of identifier,value pairs and update that list as necessary to reflect expression evaluation (see *Table* class, Appel p13)**
- **Copy files from /usr/local/class/cs415/sp99/tiger/chap1/***

First Project

- **To write interpreter without any side effects**
  - ***interpStm(), interpExp()* are coded as mutually recursive functions (see grammar, Appel p7)**
  - **A *Table* object parameter provides values with which to interpret the *Stm* or *Exp* found.**
  - ***interpStm()* returns a new *Table* object, containing any new identifier,value bindings due to side effects**
  - ***interpExp()* returns an *IntAndTable* object so as to return a *Table* plus the value of the expression**

Relevant Aspects of Java

- ***Abstract classes*** **- for organizing shared functionality (instance variables or method implementations)**
- **You cannot create an object of an abstract class type**
- **Abstract classes assume you will create subclasses of them**
- **Can also leave some methods as *abstract*, that is, without implementation**

Abstract Classes vs Interfaces

- ***Abstract classes*** **can contain method implementations and instance variables**
- ***Interfaces*** **can contain specifications of methods, but not implementations and only constant instance variables (static final)**
- **A class can inherit from more than 1 interface but only from one class (abstract or not).**

How to check object type?

- **Can simulate enumeration types with a *kind()* method that returns a different integer value in each class**
  - **Helps to identify type of tree nodes**
- ***instanceof* allows dynamic checks of the type of a Java object at run-time**

Lexical Tokens

- **Sequence of characters that form atomic pieces from which PL's are built**
  - **E.g., identifiers, reserved words, operators, delimeters**
  - **In project 1: *print*, numbers, identifiers, ( ) + * /**
- **Simple structure definable using regular expressions (or corresponding regular grammars)**
- **Instance of a token called a *lexeme***

Lexical Tokens

- **Examples of tokens and lexemes**

id	**A1**
num	**2.5**
comma	**,**

- **Tokens have associated attributes or values**
- **Scanner - part of compiler that**
  - **Finds tokens, helps in error handling (in finding next source line), finds reserved words**
  - **Handles white space. in Fortran: DO 5 I = 1, 25 versus DO 5 I = 2.5**

Regular Expressions

- We say ε is an RE representing the language which only contains the empty string
- a for a terminal a represents the language {a}
- If s,t are REs then s | t represents L(s) union L(t)
- If s,t are REs then s t represents L(s)L(t)
- If s is an RE then s* represents ε union L(s) union L(s)L(s) union L(s)L(s)L(s)… (Kleene star)
- Examples: (a^+) | b c* ={a, aa, aaa, …, b, bc, bcc, …}, a^+ means a a*

Regular Expressions

- **Shorthand notation used in JLex**

  [abcd] means a|b|c|d

  [a-z] means a|b|c|…|z

  a? means a | ε

  . means any character except newline (\n)

- **Examples of JLex specification of tokens**

  print **reserved word**

  [0-9]$^{+}$ ***num***

  [a-zA-Z]$^{+}$ ([a-zA-Z] | [0-9])* ***id***

Regular Expressions

- **May need to represent special characters**
  - **\t tab, \n newline**
- **Use “ ” to surround a string that stands for itself in a regular expression**

  ([0-9]$^{+}$ “.”[0-9]*) | ([0-9]* “.” [0-9]$^{+}$) ***real***

  (“--”[a-z]*“\n”) pattern of a comment