DEREK RAYSIDE & ECE351 STAFF

ECE351 LAB MANUAL

UNIVERSITY OF WATERLOO

ACKNOWLEDGEMENTS:

- Prof Paul Ward suggested that we look into something with VHDL to have synergy with ECE327.
- Prof Mark Aagaard, as the ECE327 instructor, consulted throughout the development of this material.
- Prof Patrick Lam generously shared his material from the last offering of ECE251.
- Zhengfang (Alex) Duanmu & Lingyun (Luke) Li [1B Elec] wrote solutions to most labs in TXL.
- Jiantong (David) Gao & Rui (Ray) Kong [3B Comp] wrote solutions to the VHDL labs in ANTLR.
- Aman Muthrej and Atulan Zaman [3A Comp] wrote solutions to the VHDL labs in Parboiled.
- Michael Thiessen [3A Comp] improved the VHDL grammar in Parboiled.
- TA's Jon Eyolfson, Vajih Montaghami, Alireza Mortezaei, Wenzhu Man, and Mohammed Hassan.
- TA Wallace Wu developed the VHDL labs.
- High school students Brian Engio and Tianyu Guo drew a number of diagrams for this manual, wrote Javadoc comments for the code, and provided helpful comments on the manual.

Contents

Compiler Concepts: desugaring, function inlining
Programming Concepts:

Tiger: repOk() methods

See the *Design Patterns* book or Wikipedia or SourceMaking.com

Tiger: Lab manual

Tiger: §4.3 + lab manual

†denotes conceptual sections

List of Figures

Reflections

Compiler Concepts: call stack, heap
Programming Concepts: version control, push, pull, merge, SSH keys, IDE, debugger, objects, pointers

Overview

0.1 *How the Labs Fit Together*

The overall structure of our VHDL synthesizer and simulator is depicted in Figure 1.

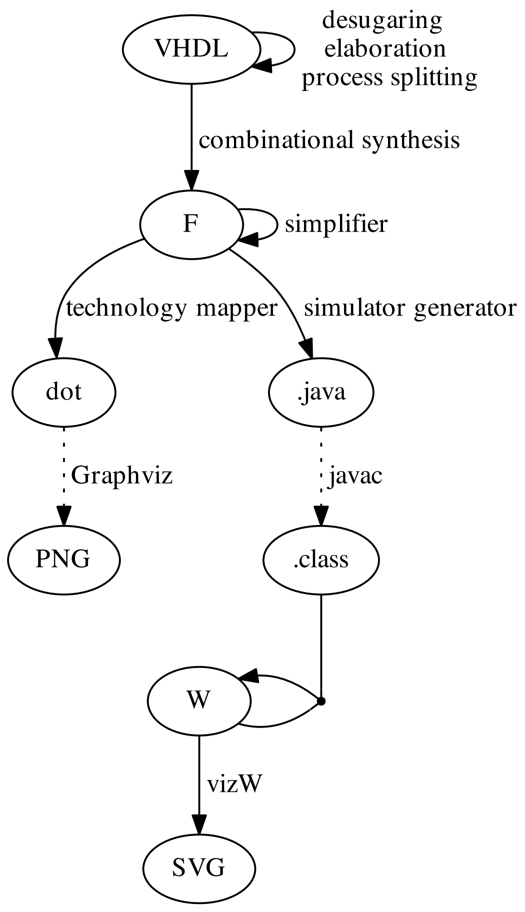


Figure 1: Overview of ECE351 Labs.
Nodes represent file types (*e.g.*, VHDL). All of these file types, with the exception of .class and PNG files, are text files.
Edges represent translators between different file types. Solid edges represent translators that we will implement in ECE351. Dotted edges represent translators provided by third-parties such as Sun/Oracle (javac) or AT&T Research (dot).
The three-part edge between .class and W nodes is intended to indicate that the .class file we generate will read a waveform (W) file as input and write a waveform (W) file as output.
Labels on edges describe the translation(s) performed.
Numbers on edges indicate the order in which we will implement those translators in ECE351. For example, the first translator that we will implement will transform waveform files to SVG files (SVG is an XML-based graphics file format).
The general direction of our work in ECE351 will be from the bottom of the figure towards the top. We will start with file types that have the simplest grammars and work towards file types with more complicated grammars.

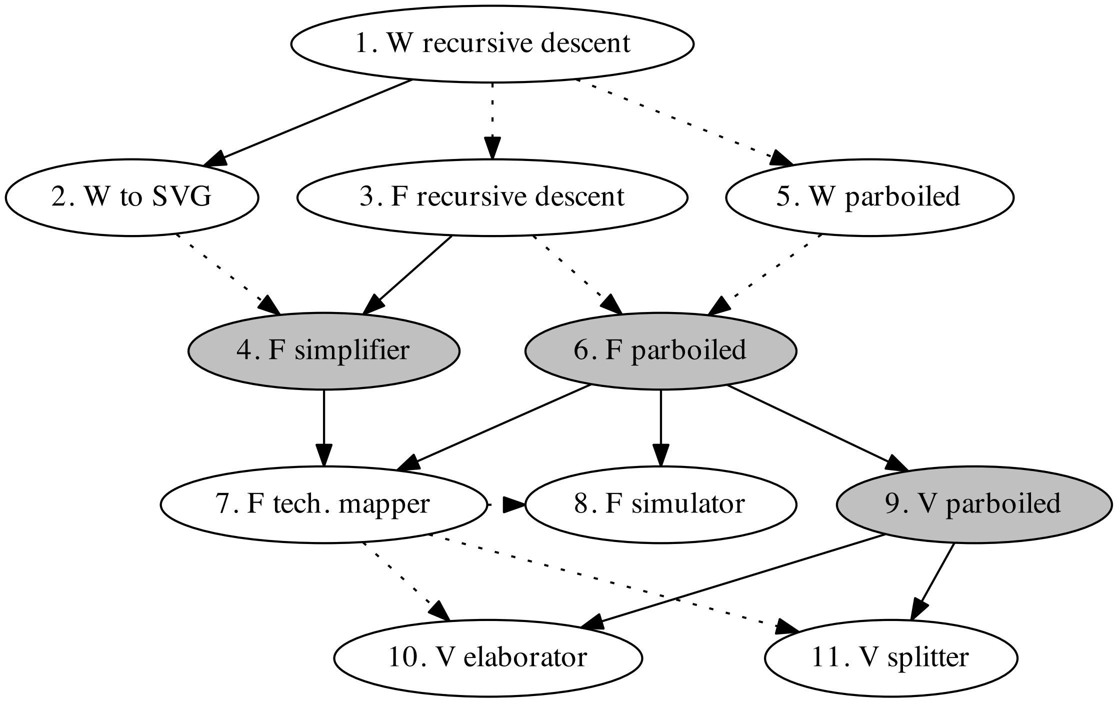


Figure 2: Lab dependencies. A solid line indicates that the code is needed to test the future lab. For example, you need to have a working implementation of lab1 in order to test lab2.

A dotted line indicates that ideas from one lab feeds into the next lab. For example, you learn the grammar of $\mathcal{W}$ in lab1 and then use that idea again in lab5. Not all of the dotted lines are shown, in order to simplify the graph.

The shaded nodes indicate labs that you need to focus on because they are used by future labs. If you must skip a lab, skip something like 2, 5, 8, 10, or 11 that are not used as much by future labs.

0.2 *Learning Progressions*

There are a variety of undergraduate compiler projects. Almost all of them involve compiling a subset of some imperative programming language to assembly. In the past Pascal was the popular language to compile, whereas these days Java is.

The ECE351 labs are different. On a superficial level, the difference is that these labs use a subset of VHDL as the input language, and produce circuit gate diagrams and simulations as outputs. The deeper difference is that our project is designed around two parallel *learning progressions*rather than around the logical structure of a compiler. This project comprises both a *programming skills progression* and a *compiler concepts progression*.

The idea of a learning progression has been used in hockey for several decades. It is recently attracting attention in educational circles. For example, suppose that the goal of the practice is to have the players skate quickly around the circles with the puck. To progress to that goal the team might start with skating around circles, then skating in a straight line with the puck, then skating around the circles slowly with the puck, and finally skating around the circles quickly with the puck. The skills are first practiced in isolation and at slower speeds, and then finally all together at high speed.

The key technical decision that enables these progressions is to restrict our subset of VHDL to combinational circuits: no loops, no timing, *etc.*. From this decision flows the design of a simple intermediate language, $\mathcal{F}$, for boolean formulas, and the design of a simple auxiliary language, $\mathcal{W}$, for boolean waveforms. The project progresses from the simplest language ($\mathcal{W}$) to the most complex (VHDL), performing parsing, transformation, and translation on each of the three languages. Repetition with increasing complexity (hopefully) leads to mastery.

#	Description	Compiler Concepts	Programming Concepts
0	Prelab	call stack, heap	version control, push, pull, merge, SSH keys, IDE, debugger, objects, pointers
1	Parsing $\mathcal{W}$ by recursive descent	regular languages, regular expressions, EBNF, recognizers, parsing, recursive descent, lexing, pretty-printing, abstract syntax tree (AST)	classes, objects, variables, aliasing, immutability, object contract, object equality, mathematical equivalence classes
2	Translating $\mathcal{W}$ to SVG (visualization)	trees, transformations, XML, SVG	object contract, DOM *vs.* SAX parser styles, call-backs, iterator design pattern
3	Parsing $\mathcal{F}$ by recursive descent	context-free grammars, LL(1) grammars, predict sets, parse trees, precedence, associativity, commutativity, program equivalence	inheritance, polymorphism, dynamic dispatch, type tests, casting, memory safety, composite design pattern, template design pattern, singleton design pattern, recursive functions, recursive structures, higher-order functions
4	Simplifying $\mathcal{F}$ programs (optimization)	intermediate languages, identity element, absorbing element, equivalence of logical formulas, term rewriting, termination, confluence, convergence	interpreter design pattern, template design pattern, representation invariants
5	Parsing $\mathcal{W}$ with a parser generator	parser generators, Parsing Expression Grammars (PEG), push-down automata	domain specific languages (DSL): internal *vs.* external, debugging generated code, stacks
6	Parsing $\mathcal{F}$ with a parser generator	parser generators, Parsing Expression Grammars (PEG), push-down automata	domain specific languages (DSL): internal *vs.* external, debugging generated code, stacks
7	Translating $\mathcal{F}$ to Graphviz (technology mapping)	common subexpression elimination	hash structures, iteration order, object identity, non-determinism, Visitor design pattern, tree traversals: preorder, postorder, inorder
8	Translating $\mathcal{F}$ to Java (circuit simulation)	program generation, name capture	
9	Parsing VHDL with a parser generator		
10	VHDL elaboration	desugaring, function inlining	
11	VHDL process splitting and translation to $\mathcal{F}$ (combinational synthesis)		
B	Translating $\mathcal{F}$ to assembly	instruction selection, register allocation	assembly, linking

* 'DP' stands for 'design pattern'; 'B' stands for bonus lab

Figure 3: Descriptions of individual labs with the compiler and programming concepts introduced in each

0.3 *How this project compares to CS241, the text book, etc.*

	ECE351	CS241	*Tiger*	CS444	MIT 6.035
Language(s)	VHDL, $\mathcal{F}$, $\mathcal{W}$	Java	Java	Java	Java
Compiler Phases					
Parsing	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
Symbol tables		$\surd$	$\surd$	$\surd$	$\surd$
Type checking		?	$\surd$	$\surd$	$\surd$
Dataflow analysis			$\circ$	?	$\surd$
Optimization	$\surd$		$\circ$	?	$\surd$
Translation	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
Assembly	$\circ$	$\surd$	$\surd$	$\surd$	$\surd$
Pedagogy					
Skills Progression	$\surd$				
Concept Progression	$\surd$				
Background	$\surd$				
Tests	$\surd$				
Workload	$\surd$	$\surd$	$\surd$	$\times 2$	$\times 2$

0.4 *Student work load*

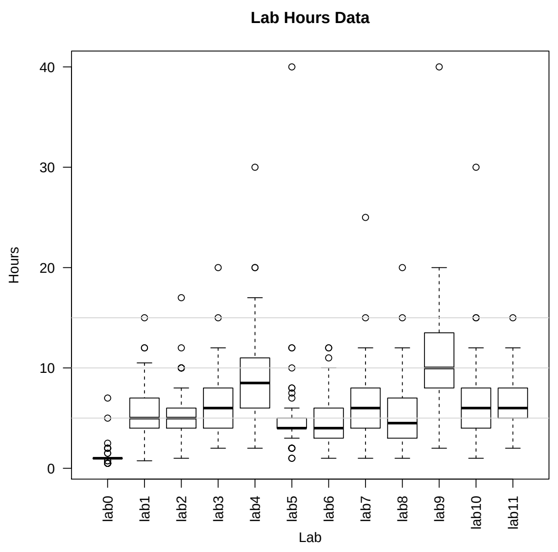


Figure 4: Student hours spent on labs in winter 2013. The target is five hours per lab. About half the students hit this target on most labs, as indicated by the bars in the middle of the boxes. On most labs, 75% of the students completed the lab within eight hours, as indicated by the top of the boxes.

Some students took much longer than eight hours. If you are one of those students, please consider taking advantage of our course collaboration policy. The point of the collaboration policy is for you to learn more in less time.

The exceptional labs that took more time were 4 and 9. This term we will be increasing the TA resources for lab 4, and also changing the way we teach lab 9, in an effort to get these numbers down.

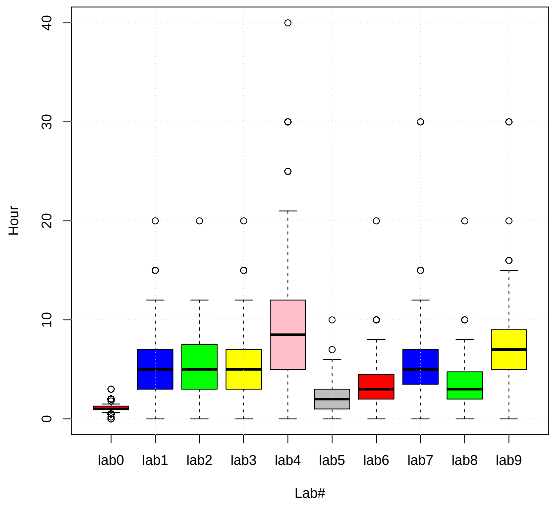


Figure 5: Student hours spent on labs in summer 2013. (Partial data to lab 9.)

0.5 How this course compares to MIT 6.035

MIT's (only) undergraduate compilers course is 6.035. It differs from ECE351 in a two important ways:

a. 6.035 is rated at 12 hours per week for 14 weeks, whereas ECE351 is rated at 10 hours per week for 12 weeks: a 48 hour nominal difference. Moreover, 6.035 makes no effort to keep the workload near the course rating, whereas ECE351 actively tracks student hours and over half the students get the work done in the rated time. So the actual difference is much larger than the nominal.

   6.035 also comes in an 18 hour per week variant.

b. 6.035 is an elective course, whereas ECE351 is a required course. Elective courses only get students who have the interest and ability; required courses get everyone. On the flipside, not every graduate of MIT EECS will know compilers: some will be totally ignorant. At UW we guarantee a minimum level of quality in our graduates (this approach is also required for CEAB accreditation).

0.6 Where do I learn more?

If you are interested in learning more about compilers at UW your next step is CS444. If you take ECE351 and CS444 then you will know more than if you took MIT 6.035. CS462 covers the theoretical end of formal languages and parsing, although this knowledge won't greatly improve your practical skills. In the graduate curriculum CS644 and CS744 are offered on a semi-regular basis. There are also a number of graduate courses in both ECE and CS on program analysis that are offered irregularly.

0.7 Full-Adder Circuit Example

```
entity full_adder is port (
    A, B, Cin: in bit;
    S, Cout: out bit
);
end full_adder;

architecture full_adder_arch of full_adder is
begin
    S <= (A xor B) xor Cin;
    Cout <= ((A xor B) and Cin) or (A and B);
end full_adder_arch;
```

Figure 6: Source code for full adder circuit (VHDL)

```
A: 0 1 0 1 0 1 0 1;
B: 0 0 1 1 0 0 1 1;
Cin: 0 0 0 0 1 1 1 1;
```

Figure 7: Input waveform for full adder ($\mathcal{W}$)

```
S <= ((not (((not A) and B) or ((not B) and A))) and Cin) or
        ((((not A) and B) or ((not B) and A)) and (not Cin));
Cout <= ((((not A) and B) or ((not B) and A)) and Cin)
        or (A and B);
```

Figure 8: Boolean formulas for full adder ($\mathcal{F}$ generated from source code in Figure 6)

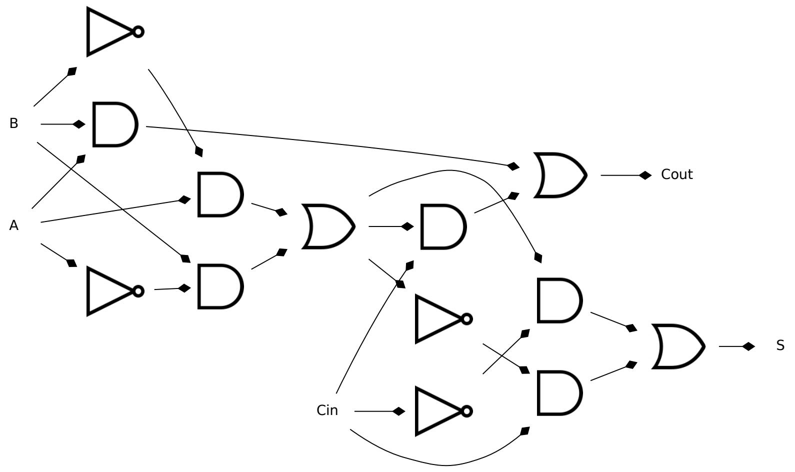


Figure 9: Gates for full adder (generated from formulas in Figure 8)

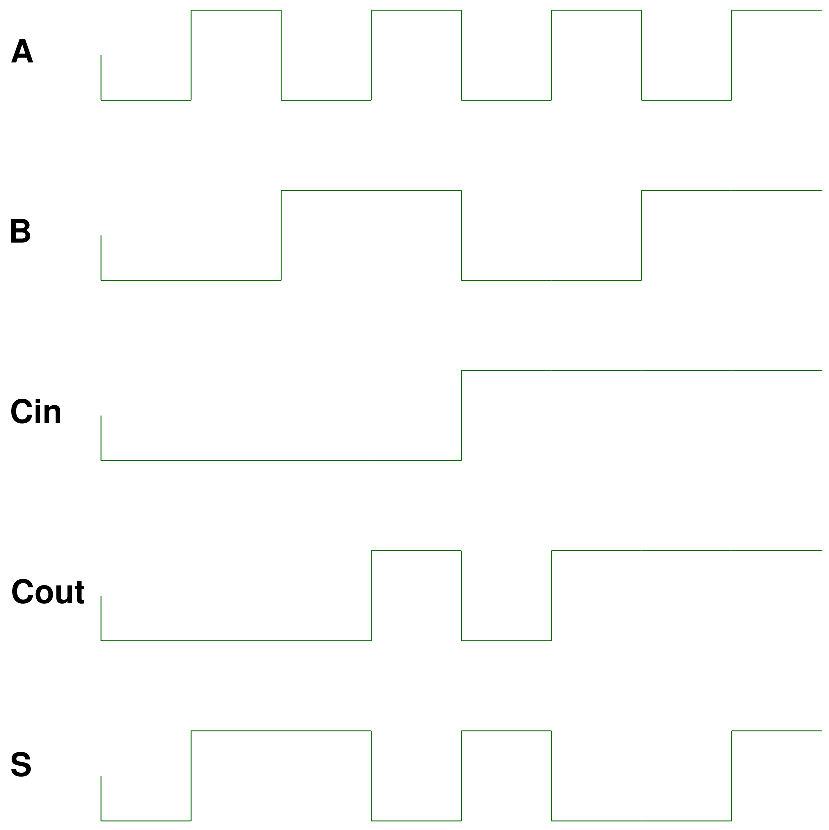


Figure 10: Input and output waveforms for full adder (generated from formulas in Figure 8 and input waveform in Figure 7)

0.8 Pre-Lab: Computing Environment

0.8.1 Our Environment

Follow the instructions at ecegit.uwaterloo.ca/documentation/ and ensure you can authenticate properly. There are instructions for installing `git` (the version control software we're using) and a link to git references as well. Note that git is already installed in the computers in E2 2363.

0.8.2 Getting Your Code

All of the code and materials are hosted on ecegit.uwaterloo.ca. Follow these steps to set up your repository.

`TERM` is the term number, which you can find at http://www.adm.uwaterloo.ca/infocour/CIR/SA/under.html. It is four digits. The first digit is always 1. The second two digits are the year. The last digit is the month the term starts on. So 1145 indicates spring term of 2014 (starts in May, the fifth month).

- ☐ `mkdir ~/git`
- ☐ `cd ~/git`
- ☐ `git clone git@ecegit.uwaterloo.ca:ece351/TERM/USERNAME/labs ece351-labs`

replace `USERNAME` with your username

- ☐ `cd ece351-labs`
- ☐ `git submodule init`
- ☐ `git submodule update`
- ☐ `git remote add skeleton git@ecegit.uwaterloo.ca:ece351/TERM/skeleton`

0.8.3 Updating Your Repository

If there are updates, assuming you're back on the `master` branch, type the following command to merge in the changes:

- ☐ `cd ~/git/ece351-labs`
- ☐ `git pull`
- ☐ `git pull skeleton master`
- ☐ `git submodule update`

If there are any conflicts type `git status` to check which files you need to modify. Modify the file(s) and keep whatever changes you want. Afterwards commit your conflict resolution for the merge. Clean merges will create a commit for you.

0.8.4 Committing Changes

Commit after every logical work unit.

When you have changes you want to add to the repository (they should be visible when you type `git status`) you need to commit them. If status shows files that are modified you can do `git commit -am "Descriptive message here"` to add them to your repository. This should be the type of commit you use most of the time during the labs.

If you want to read up on git here are the most important parts you'll need to know:

- http://git-scm.com/book/en/Getting-Started-Git-Basics discusses working directory / staging area / git repository
- http://git-scm.com/book/en/Git-Basics-Getting-a-Git-Repository how to use it locally
- http://git-scm.com/book/en/Git-Branching-What-a-Branch-Is basics on branching
- http://git-scm.com/book/en/Git-Branching-Basic-Branching-and-Merging merging
- http://git-scm.com/book/en/Git-Basics-Working-with-Remotes working with servers, we're using two remotes (`origin` for your changes and `skeleton` for staff changes)
- http://git-scm.com/book/en/Git-Tools-Submodules you won't need this yet, but when we start testing you'll need this

0.8.5 Uploading Your Repository

Push at the end of every work session.

When you want to upload your commits to ecegit simply type the following:

☐ `git push`

0.8.6 Picturing Git

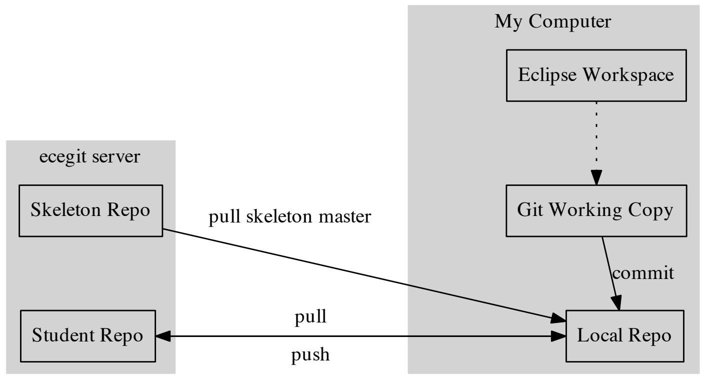


Figure 11: Relationship between local repository, student server repository, and skeleton repository

0.8.7 Eclipse on your computer

Eclipse is already installed on the lab computers. These instructions are for installing it on your computer, if you choose to do so.

Q:\eng\ece\util\eclipse.bat

☐ Download Eclipse Classic 4.2.2 from http://www.eclipse.org/downloads/packages/eclipse-classic-421/junosr1

☐ Launch Eclipse

☐ Click **File** then **Import** or pres Alt-f then i

☐ Open the **General** folder

☐ Select **Existing Projects into Workspace**

☐ Click **Next**

☐ Set **Select root directory:** `~/git/ece351-labs` (or wherever your repository is

☐ Make sure ece351 is checked under **Projects**

☐ Make sure **Copy projects into workspace** is **unchecked**

☐ Click **Finish**

0.8.8 Configuring Eclipse

JUnit Launcher to enable assertions:

☐ Go to **Window / Preferences / Java / JUnit**
☐ Check **Add -ea to VM arguments for new launch configurations**
☐ Click **Apply** and close the window

Switch package presentation mode to hierarchical:

☐ Click the downwards pointing triangle icon in the **Package Explorer**
☐ Select **Package Presentation / Heirarchical** on the menu

0.8.9 Standard Java Data Structures

Open java.util.List in Eclipse's Type Hierarchy View. What are the subclasses of List in the java.util package? What is the difference between them?

To open a type in Type Hierarchy view select *Open Type in Hierarchy* from the *Navigate* menu, or press Shift+Ctrl+H, and then key in the name of the type.

Open java.util.Set in Eclipse's Type Hierarchy View. What are the subclasses of Set in the java.util package? What are the differences between them?

Open java.util.Map in Eclipse's Type Hierarchy View. What are the subclasses of Map in the java.util package? What are the differences between them?

0.8.10 Pre-Lab Exercise

We created an update for you to merge into yours, follow the above instructions to get it. Afterwards there should be `meta` directories in your repository. Edit `meta/hours.txt` with how many hours it took for lab0.[1]

Here is a checklist:

- ☐ Got updates from the skeleton
- ☐ Merged updates into your master branch
- ☐ Run TestPrelab (testEclipseJDKConfiguration and testJUnitConfiguration should pass, the rest should fail and you should read the reason why)
- ☐ Run TestImports (should pass)
- ☐ Run build.xml (this is the marking script)
- ☐ Right-click on a single test to run it individually
- ☐ Familiarize yourself with the Eclipse debugger: breakpoints, watches, *etc.*
- ☐ Edit `meta/hours.txt`
- ☐ Uploaded your changes

[1] lab0 is this prelab exercise, including any time you spend installing software on your own computer.

Common Problem 1: Second TestPrelab test fails. If you are on the machines in E2 2363, you may find that the second test in TestPrelab does not pass. In order to fix this, do the following. Click Window and go to Preferences. In the tree view select Java, then Installed JREs and click Add. Hit Next (Standard VM should be selected already), then Directory, navigate to C:\Program Files\Java\jdk, or something similar, and hit Finish. Finally, check the jdk in the Installed JREs pane and click OK.

Common Problem 2: testJUnitConfiguration fails, but you have followed the steps above in §0.8.8. Probably you tried to run TestPrelab *before* following the steps from §0.8.8. The steps from §0.8.8 only adds '−`ea`' to *new* run configurations; it does not change existing run configurations. You now need to change the TestPrelab run configuration individually so that it includes '−`ea`' in the VM arguments.

0.9 How to do these labs

The lab manual will tell you what files you need to edit, what libraries you should use, and what tests to run. The lab manual will also explain what you need to do at each point in the code, and the background concepts you need to understand the lab. Most of the dagger (†) sections in the lab manual are to explain these background concepts. Every place in the skeleton code that you need to edit is marked by both a `Todo351Exception` and a `TODO` marker. We will discuss the next week's lab in class every Friday.

Despite this clear and explicit instruction, some students have difficulty getting started on these labs. Why? What is missing from the instructions described above? *The lab manual doesn't tell you the order in which you should edit the files.* There is an important reason for this: execution order is the wrong way to think about developing object-oriented software (most of the time). The right way to think about object-oriented software is to focus on developing cohesive and extendible modules (classes). In this way object-oriented programming (in the large) is mentally quite different from procedural programming (in the small). In procedural programming (in the small) one thinks primarily about the order in which the steps are performed.

Now don't misunderstand this to mean that execution order doesn't matter: it does. It's just that execution order is generally a separate design concern from modularity and extensibility. The best

In ECE250 you did object-oriented programming in the small. That is, you defined structures that spanned one or two classes and operations on those structures that spanned one or two methods. The programs you worked on were perhaps several hundred lines long.

In ECE351 you will work on a code base that is over 8,000 lines of code, of which you will write about 1000 lines, at an average rate of about 100 lines per week. The structures we will work with are defined across dozens of classes, and the operations on those structures are similarly defined across dozens of methods. This is the first time in the ECE curriculum that you are exposed to *programming in the large*. At this scale, modularity and code structure really matter.

By the standards of modern industrial software development, 8,000+ lines is just approaching medium sized. The code structuring ideas you will learn in these labs can take you up to maybe 100,000 lines: beyond that you will need new ideas and modularity mechanisms.

way to understand the execution order of a large object-oriented program is to run it and observe it in the debugger or via some other tracing mechanism (*e.g.*, printf).

If you want to figure out what code you should edit first, run the test harness and see where an exception is thrown. Fix that. Then run again, get another exception, *etc.*. The 'fix' step might be quick and easy, or it might require reading several pages of the lab manual to understand what needs to be done at that particular point. Remember, the lab manual describes everything that needs to be done and where it needs to be done, it just doesn't describe the order in which you should do it. Aligning your editing order with the program execution order is one way to guide your work that will help you build an understanding of the code.

Thinking on your own to develop an understanding of the skeleton code is an important part of these labs. I promise you that it takes less time and effort to study 8,000+ lines of code than to write it from scratch.

0.10 Metadata

Your workspace has a directory named *meta* that contains the following three files in which you can describe a few things about how your work on the lab went.

collaboration.txt To record your collaborators. Each line is a triple of lab number, collaboration role, and userid. Legal values for collaboration roles are: verbal, partner, mentor, protege. The role field describes the role of the *other* person, so lab2 mentor jsmith says that J Smith was your mentor for lab2. Similarly, lab3 protege jsmith says that J Smith was your protégé for lab3 (*i.e.*, you were his mentor). Both parties are required to report collaborations. If you collaborated with more than one person on a lab then you should put multiple lines into this file for that lab: one line for each collaborator on each lab.

You must edit this file for every lab, even if all of your collaborations were just verbal.

hours.txt Estimate of the hours you worked on each lab. This file will have a line for each lab like so: lab1 5 (indicating five hours spent on lab 1). For pre-lab / computing environment time, use lab0.

These data will be used solely for the staff to assess the difficulty of the labs. This assessment will be made in aggregate, and not on an individual basis. These data will not be used to assess your grade. However, we will not mark your lab until you report an estimate of your hours.

reflections.txt A free-form text file where you can give feedback to the course staff about what was interesting or difficult or annoying in the labs. We will use this feedback to improve the labs for future offerings of this course — and for this offering if possible. If you write something interesting here it will count towards your class participation mark.

0.11 I think I found a bug in the skeleton code

You are welcome to do what you like with the skeleton code. Our recommendation if you want to make a change is the following:

a. Report the change you want to make (in the forum, or to a staff member). Preferably by generating a patch.

b. We tell you that the change is misguided and you really want to do something else.

c. We say thanks, add that to your participation.txt and we patch the skeleton code with the change so everyone can use it.

If you make the change yourself without reporting it then you lose out on class participation points, and if someone else reports the change and we go to apply a patch the patch will fail on your code. This might or might not end up being a problem for you.

0.12 I want to change the skeleton code for my own usage

You may change the skeleton code for your own usage. Perhaps you have a better idea of how to write something. If you make such changes, be careful to not create more problems than you solve. Some things to look out for include:

- Breaking the JUnit test harnesses.
- Breaking some other code that depends on the code you are changing.
- Messing up one of the design patterns that are an explicit part of what you are supposed to be learning.
- Changing something from immutable to mutable because you do not want to learn to work with immutable data. First, this is depriving yourself of one of the important lessons of the labs. Second, you will likely be introducing some bugs that will be very difficult to fix later. Immutable data is a good engineering practice that helps you avoid many classes of difficult to diagnose and fix bugs.

Algorithmic changes are usually safe, because the algorithmic parts of this code are usually encapsulated.

0.13 Object-Oriented Programming†

You will need to understand object-oriented programming,[2] including classes, objects, methods, fields, local variables, inheritance / subclassing / subtyping, polymorphism / dynamic dispatch, and method overloading. You will also need to understand some design patterns,[3] such as *iterator*, *composite*, and *visitor*. We will make some effort to cover these things, but they are in some sense really background material for this course that you are expected to know.

[2] B. Eckel. *Thinking in Java*. Prentice-Hall, 2002. http://www.mindview.net/Books/TIJ/

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass., 1995

```
class B {
    String x;
    String foo() {
        String y = x + "_";
        return y;
    }
}
class D extends B {
    String foo() { return x; }
}
```

- B is a *base class* or *super-class*
- D is a *derived class* or *sub-class*
- D *extends/derives/subclasses* B
- x is a *field* or *instance variable*
- y is a *local variable*
- foo is a *method* (a *virtual method* in C++ terms)
- the definition of foo in D *overrides* the definition of foo in B

Figure 12: Basic terminology of OOP

0.13.1 Visualizing Objects and Pointers

It is absolutely vital that you understand objects and pointers, both for your own programming in the labs for this course and elsewhere, and to understand the subject material of this course. A great way to solidify these concepts is through an appropriate visualization system such as Jeliot (for Java) and PythonTutor.com (for Python). The fundamental concepts of objects and pointers are the same regardless of language: Java, Python, C#, C++, *etc.* You will notice that Jeliot and PythonTutor.com use very similar visualizations.

http://www.cs.joensuu.fi/jeliot/
PythonTutor.com

There is also a nice set of 'learning objects' (lessons) for learning Java with the Jeliot visualization system.

http://cnx.org/content/col10915/latest/

0.14 Testing†

Program testing can be used to show the presence of bugs, but never to show their absence! — Edsger W. Dijstra

Just because your code passes all of the tests does not mean it is correct: there could be some as of yet unidentified test that it does not pass. Worse, that not-yet-identified test might occur during some lab later in the term.

This might be the first course in which you have to write a 'real' program: that is, a non-trivial program that you will have to depend on in the future. First, the labs are non-trivial: The total size of the code for this course is about 8500 lines, of which you will have to write about 1500, and roughly 7000 will be provided for you. Second, the labs are cumulative: Each week you will run some labs that you wrote in the past. At the end of the term you will run all of them together. So testing will be important.

Test inputs may be generated either *manually* or automatically with a testing tool. There are two main approaches used by tools: *systematic* and *random*. Because the space of possible inputs is large, possibly infinite, systematic approaches tend to generate all small inputs and no big inputs. Random approaches will generate a mixture of small and large inputs. Programmers can use human insight to generate interesting inputs that might be missed by automated tools.

The Korat tool generates test inputs systematically based on representation invariants.

The Randoop tool generates inputs randomly.

There are different strategies for evaluating whether a test passed or failed. One is to check the computed result against a known answer. In other words, the test suite consists of a set of known input/output pairs.

Another strategy is to check general properties of functions. For example, that a function never returns null. An advantage of this property-based approach is that one only needs the test inputs — the corresponding outputs do not need to known. Some properties that we will be interested in for this course include:

reflexive	x.equals(x)
symmetric	x.equals(y) $\Rightarrow$ y.equals(x)
transitive	x.equals(y) and y.equals(z) $\Rightarrow$ x.equals(z)
antisymmetric	$x \leq y$ and $y \leq x \Rightarrow x = y$
total	$x \leq y$ or $y \leq x$
idempotent	$f(x) = f(f(x))$
invertible	$f'(f(x)) = x$

Some functions do not have any of these properties. But if we expect a function to have one or more of these properties then it is a good idea to test for that property explicitly.

In particular, the first three properties define a mathematical *equivalence relation*. We expect the equals() method in Java to represent a mathematical equivalence relation.

A *total order* is defined to be transitive, antisymmetric, and total. The integers, for example, are totally ordered. We expect the compareTo() method in Java to represent a total order.

A good test suite will contain some of everything mentioned above: test inputs that are generated manually and automatically, both systematically and randomly; evaluations that look at specific input/output pairs and at general properties.

In a previous offering of this course the staff provided tests that only looked at general properties and none that examined specific input output pairs. It turned out that students could write code that had the general property without actually computing anything useful. They became unhappy when they discovered that their code that passed the staff-provided tests did not actually work when they wanted to run it on a future lab.

0.15 *Phases of Compilation*†

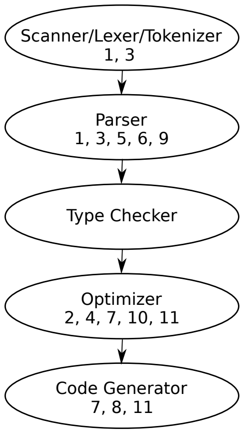


Figure 13: Phases of compilation and the labs in which we see them. The scanner, parser, and type checker are considered the *front end*, whereas the optimizer and code generator are considered the *back end* of the compiler.

0.16 Engineers and Other Educated Persons

There are a number of important general skills that all educated persons, including engineers, should possess. The first two years of engineering education need to focus on technical details in isolation in order for you to have the technical competency to tackle more interesting things. This focus on minutiæ sometimes comes at the cost of limited growth in these larger skills. In this course you will not only learn lots of tiny technical details, but you will also need to develop some of the larger skills that all educated persons should possess.

Simplicity and elegance are unpopular because they require hard work and discipline to achieve and education to be appreciated.
– Edsgar W. Dijkstra, 1997

The ability to quickly find relevant information. Information is more accessible now than at any previous point in human history. There are a wide variety of sources available to you: the course notes, the lab manual, old exams, the skeleton code, the recommended text books, other text books, lecture notes and slides and videos from other professors, *etc.* You should be facile in using all of these sources to find what you are looking for.

Books, in particular, have helpful features for navigating them, such as the table of contents, the index, and the preface or introduction. You should know how to use these features.

Fools ignore complexity; pragmatists suffer it; experts avoid it; geniuses remove it.
— Alan Perlis

The ability to quickly assess what information is relevant. As old-time engineers used to say, there is always more heat than light. Learn to see the light without being overwhelmed by the heat.

When doctors, lawyers, accountants, and other professionals assess a case study problem, the first order of business is to discern what the relevant facts and issues are. Those professions explicitly train their people to separate the wheat from the chaff. Engineering education, especially in the first two years, is often guilty of spoon-feeding its students with only and all of the relevant information, and hence developing a sense of intellectual complacency and entitlement in its students.

For example, you should be able to take a list of topics to be covered from the course outline and be able to use that to determine which sections of the text book are relevant and which questions from old exams are applicable.

THE ABILITY TO ACCURATELY ASSESS THE CREDIBILITY OF A SOURCE. Information comes from a variety of sources. Some of them are more credible than others. An educated person knows, for example, that a report from the Transportation Safety Board of Canada is more likely to be accurate than a newspaper report — more likely, but not infallibly so.

There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies. And the other way is to make it so complicated that there are no obvious deficiencies.
— C.A.R. Hoare, 1982
Turing Award Speech

THE ABILITY TO MANAGE LARGE AMOUNTS OF COMPLEX, INTERCONNECTED DETAILS. The human body is a complex system that doctors work with. The law is a complex system that lawyers work with. Engineers work with complex socio-technical systems.

In the first two years of engineering education you typically encounter only small text book problems in order to gain understanding of specific technical concepts in isolation. The labs for this course might be the first time in your engineering education where you have had to work with a non-toy system. Our skeleton code is still small by industrial standards, but it might be an order of magnitude greater than what you have worked with in your previous courses. Learning to manage this volume of complexity and interdependency is an essential part of your professional education.

The total code base that we work with in this course is about 8,000 lines. We provide about 7,000 lines of that to you — which you are expected to understand while developing the remaining 1,000 lines.

For reference, the source code for Microsoft's Windows operating system is somewhere around 25 *million* lines of code. The source code for ATI's video card driver is about *60 million lines* — more than double the Windows operating system. Modern chip designs also get into millions of lines of code.

THE ABILITY TO THINK DEEP THOUGHTS. In elementary school you learned to multiply positive integers. In middle school you learned to multiply all integers (including negative ones). In high school you learned to multiply matrices, perhaps in two different ways: cross-product and dot-product. In first year you learned to write programs that multiply matrices. In this course you will learn a bit about how to design and implement programming languages in which someone might write a program to multiply matrices. You could go on in pure math and logic courses to learn more about multiplication. It's still the multiplication of your childhood, but your understanding of it is much deeper for the twenty years of study that you have devoted to it. Understanding one or a few things in some depth like this hopefully cultivates in you the ability to think deeper thoughts in other domains and circumstances.

Back where I come from we have universities — seats of great learning — where men go to become great thinkers. And when they come out, they think deep thoughts, and with no more brains than you have.
— The Wizard of Oz, 1939

Of course, in the modern world, women now go to (and graduate from) universities in greater numbers than men.

For example, Gödel's first incompleteness theorem shows that any formal logic that includes multiplication can express propositions that we know to be true but which cannot be proven within that logic. A result that rocked the world in 1931, and was an important intellectual precursor to Turing's proof of The Halting Problem in 1936.

Lab 1
Recursive Descent Parsing of $\mathcal{W}$

Compiler Concepts: regular languages, regular expressions, EBNF, recognizers, parsing, recursive descent, lexing, pretty-printing, abstract syntax tree (AST)

Programming Concepts: classes, objects, variables, aliasing, immutability, object contract, object equality, mathematical equivalence classes

We consider that the input and outputs of a circuit are a set of waveforms. We use the $\mathcal{W}$ language for expressing waveforms in text files. Figure 1.1 shows an example $\mathcal{W}$ file and Figure 1.2 gives the grammar for $\mathcal{W}$.

```
A: 0 1 0 1 0 1 ;
B: 1 0 1 0 1 0 ;
OR: 1 1 1 1 1 1 ;
```

Figure 1.1: Example waveform file for an OR gate. The input pins are named A and B, and the output pin is named OR. Our VHDL simulator will read a $\mathcal{W}$ file with lines A and B and will produce a $\mathcal{W}$ file with all three lines.

Program	$\rightarrow$	(*Waveform*)$^+$
Waveform	$\rightarrow$	*Id* ':' *Bits* ';'
Id	$\rightarrow$	*Char* (*Char* | *Digit* | '_')*
Char	$\rightarrow$	`[A-Za-z]`
Digit	$\rightarrow$	`[0-9]`
Bits	$\rightarrow$	('0' | '1')$^+$

Figure 1.2: Grammar for $\mathcal{W}$ in EBNF. By convention we will call the top production *Program*.

1.1 Write a regular expression recognizer for $\mathcal{W}$

Sources:
ece351.w.regex.TestWRegexSimple
ece351.w.regex.TestWRegexAccept
ece351.w.regex.TestWRegexReject

A *recognizer* is a program that *accepts* or *rejects* a string input based on whether that string is a valid sentence in some language. A recognizer for $\mathcal{W}$ will accept or reject a file if that file is a legal $\mathcal{W}$ 'sentence'/'program'.

$\mathcal{W}$ is a *regular* language. *Regular* is a technical term here that describes the complexity of the grammar of $\mathcal{W}$. Regular languages are the simplest kind of languages that we will consider. The grammar of a regular language can be described by a *regular expression*.

The website regexper.com will draw finite-state machine diagrams for your regular expressions. The website regexpal.com lets you test out your regular expressions. There are many other similar websites.

Your first task is to write a regular expression describing the grammar of $\mathcal{W}$. Your final answer should be stored in the field TestWRegexAccept.REGEX. The file TestWRegexSimple contains methods testr1 through testr9 that guide you through building up this regular expression in a systematic way.

A challenge that you will face is that the $\mathcal{W}$ grammar in Figure 1.2 does not explicitly specify whitespace, whereas your regular expression will have to explicitly specify the whitespace. Grammars given in EBNF usually implicitly assume some *lexical* specification that describes how characters in the input string are to be grouped together to form *tokens*. Usually any amount of whitespace may occur between tokens.

1.2 *Write a recursive descent recognizer for $\mathcal{W}$* †

Sources:
ece351.w.rdescent.WRecursiveDescentRecognizer
Libraries:
ece351.util.Lexer
Tests:
ece351.w.rdescent.TestWRDRecognizerAccept
ece351.w.rdescent.TestWRDRecognizerReject

Recursive descent is a style of writing parsers (or recognizers) by hand (*i.e.*, without using a parser generator tool). In this style we make a method for each non-terminal in the grammar. For recognizers these methods return void.

The bodies of these non-terminal methods consume the input one token at a time. Kleene stars (*) or plusses (+) become loops. Alternation bars (|) become conditionals (if statements).

Execution of these non-terminal methods starts at the 'top' of the grammar. This is what the term *descent* in *recursive descent* refers to. The term *recursive* in *recursive descent* simply means that the various non-terminal methods may call each other.

By convention we will name the top production of our grammars *Program*, and so execution of our hand-written recursive descent parsers and recognizers will start in a method called `program`.

Your $\mathcal{W}$ recognizer code will make use of the `Lexer` library class that we provide. This class *tokenizes* the input string: *i.e.*, strips out the whitespace and groups the input characters into chunks called *tokens*. The lexer provides a number of convenience methods, including methods for recognizing identifiers, so you won't need to write an `id()` method in your recognizer.

The lexer has two main kinds of operations: *inspect* the next token to see what it looks like, and *consume* the next token. The inspect methods are commonly used in the tests of loops and conditionals. The consume methods are commonly used in regular statements.

1.3 Write a pretty-printer for $\mathcal{W}$

Pretty-printing is the opposite of parsing. Parsing is the process of constructing an *abstract syntax tree* (AST) from a string input. Pretty-printing is the process of producing a string from an AST.

For this step you will write the toString() methods for the $\mathcal{W}$ AST classes. These methods will be tested in the next step. Note that these toString() methods just return a string: they do not actually print to the console nor to a file. *Pretty-printing* is the name for the inverse of parsing, and does not necessarily involve actually printing the result to an output stream. Whereas parsing constructs a tree from a string, pretty-printing produces a string from a tree.

Sources:
ece351.w.ast.WProgram
ece351.w.ast.Waveform
Libraries:
java.lang.String
System.getProperty("line.separator")
org.parboiled.common.ImmutableList

Once we have a function and its inverse we can test that $f'(f(x)) = x$ for any input x.

1.4 Write a recursive descent parser for $\mathcal{W}$

A parser reads a string input and constructs a tree output. Specifically, an *abstract syntax tree* (AST).

To write your recursive descent parser for $\mathcal{W}$ start by copying over the code from your recursive descent recognizer. The parser will be a superset of this code. The difference is that the recognizer discards the input whereas the parser will build up the AST based on the input. The AST classes are WProgram and Waveform.

The test performed here is to parse the input, pretty-print it, re-parse the pretty-printed output, and then compare the AST's from the two parses to see that they are the same.

Sources:
ece351.w.rdescent.WRecursiveDescentParser
Libraries:
ece351.w.ast.WProgram
ece351.w.ast.Waveform
ece351.util.Lexer
org.parboiled.common.ImmutableList
Tests:
ece351.w.rdescent.TestWRDParserBasic
ece351.w.rdescent.TestWRDParserAccept
ece351.w.rdescent.TestWRDParserReject

1.5 Pointers, Aliasing, and Immutability†

One of the most important design decisions to be made in a program is which data will be *mutable*. Mutability is a powerful feature that can introduce all kinds of difficult bugs if used carelessly. Two classic examples of where to use mutability and where not to are bank accounts and integers, respectively. The whole point of a bank account object is that the balance can change — that it can be mutated. An integer object, by contrast, is representing an unchanging mathematical abstraction. A good design question to ask yourself is whether the objects you are trying to model in your program represent things in the material world (that change, as all material things do); or are mathematical abstractions (which do not change). In this course we are dealing primarily with mathematical abstractions, and so most of our objects will be immutable.

Aliasing occurs when there is more than one variable pointing to an object: *i.e.*, the object has more than one name. Aliasing is not a problem for immutable objects: because the immutable object can-

Mutable means 'can be changed' or 'can be mutated'. *Immutable* means 'unchangeable'.

How do decide if an object should be mutable?

A favourite example of aliasing for philosophers is that the terms 'the morning star' and 'the evening star' both refer to the planet Venus.

Aliasing is common for people in real life and in literature. For example, in Dostoevsky's novel *The Brothers Karamazov*, each of the main characters have several nicknames. The youngest brother, Alexei Fyodorovich Karamazov, is also known as: Alyosha, Alyoshka, Alyoshenka, Alyoshechka, Alxeichick, Lyosha, and Lyoshenka. Nicknames, formal titles, and the like, add depth to human experience. But just as this depth can make it more difficult to read novels, it can also make it more difficult to read programs.

not change, it can be shared freely. Aliasing can introduce subtle bugs when mixed with mutable data though. Consider two people X and Y who share a bank account. At the beginning of the day the balance is \$100. Both people head out their separate ways to separate stores. X makes a purchase for \$50. A minute later, at a different store, Y attempts to make a purchase for \$75 but is denied because the account no longer has sufficient funds. In this real life example it is fairly easy for person Y to debug what has happened. But in a large program it can be very difficult to determine where aliases exist. And, of course, discovering the alias is just the first step towards fixing the program so that the problem does not occur during the next execution.

Try PythonTutor.com to see some nice visualizations of code execution that might help you understand variables, objects, and aliasing.

In some programming languages, such as Python, all objects are mutable; whereas in other programming languages, such as Haskell, all objects are immutable. Languages like Java allow you to choose which objects are mutable and which are immutable. Aliasing occurs in all programming languages.

Introductory object-oriented programming is often taught in a style where all objects are mutable and there is no aliasing. This style doesn't work for large programs. Large programs have aliasing — and probably aliasing across threads. Immutability is one simple technique to make aliasing a non-issue. Other solutions include complex sharing, locking, or usage protocols. These alternatives all introduce new possibilities for new kinds of subtle bugs: *e.g.*, forgetting to follow the protocol and lock or unlock the object. Immutability might, in some circumstances, introduce minor inconvenience in the short term, but it eliminates entire classes of bugs rather than introducing new possibilities for bugs.

All of our AST objects will be immutable. We will generally use mutable data only in temporary local variables, and will try to avoid storing mutable values into the heap.

The examples in Figure 1.3 illustrates the interaction of aliasing and mutability and immutability, respectively.

```
List list = new LinkedList();
List alias = list;
list.add("hello");
alias.add("world");
System.out.println(list); // 1
```

```
ImmutableList list = ImmutableList.of();
ImmutableList alias = list;
list.append("hello");
alias.append("world");
System.out.println(list); // 2
list = list.append("hello");
alias = alias.append("world");
System.out.println(list); // 3
```

Figure 1.3: Aliasing and mutable *vs.* immutable data. What output is printed? What does the heap look like? Which variables point to which objects? What is the difference between the add() and append() methods?

The LinkedList class is part of the JDK. The ImmutableList class comes with the Parboiled parser generator that we will use in later labs.

1.6 *Assignment*†

Assignment is the feature of programming languages that lets you change what a variable name means. It is one of the distinguishing characteristics of imperative programming languages. Consider the code listing in Figure 1.4: the variable x points to 'foo' first, and then we re-assign it to 'bar'.

```
String x;
x = "foo"; // assign String "foo" to variable x
x = "bar"; // re—assign variable x with String "bar"
```

Figure 1.4: Assignment and re-assignment of *x*

Assignment makes it more difficult to analyze and transform programs, as illustrated in Figure 1.5. The issue is not that we give a value a name, but that we then later change what that name means by re-assigning it to some other value. In other words, re-assignment is what introduces potential problems. Re-assignment is the basis of mutation: all mutation involves changing the value associated with some variable name (local variable, field, method parameter).

(a) program that can be optimized

```
a = (x + y) + z;
b = (x + y) * z;
```

(b) optimization of *(a)*

```
t = x + y;
a = t + z;
b = t * z;
```

(c) program that cannot be optimized

```
a = (x + y) + z;
x = 7;
b = (x + y) * z;
```

Figure 1.5: Assignment interferes with analysis and optimization. Program *(c)* cannot be optimized because *x* has been re-assigned, so the expression x+y on the last line will have a different value than the expression x+y on the first line.

Assignment and aliasing are duals of each other. Assignment allows one name to have multiple meanings (values). Aliasing is when one meaning has multiple names. Both of these language features can be very useful, but they both come at some cost: they make the program harder to understand, make it more likely to have bugs, make it more difficult to debug, and make it harder to analyze and optimize. The cost of these features is born both by the end programmer and by the compiler engineer.

In the lab skeleton code we have made an effort to mitigate the complexities of these features by following some guidelines:

- All AST objects are immutable so that they may be safely aliased and confidently reasoned about. Compilers typically have many phases that transform the AST. If these transformations actually mutate the original AST, instead of constructing a new AST, then it can be more difficult to debug.
- Wherever possible, fields, local variables, and method parameters are declared to be 'final' to prevent them from being re-assigned. The keyword 'final' in Java requires the compiler to prove that the variable is assigned exactly once on every control-flow path.

1.7 *Object Diagram of Example $\mathcal{W}$* AST

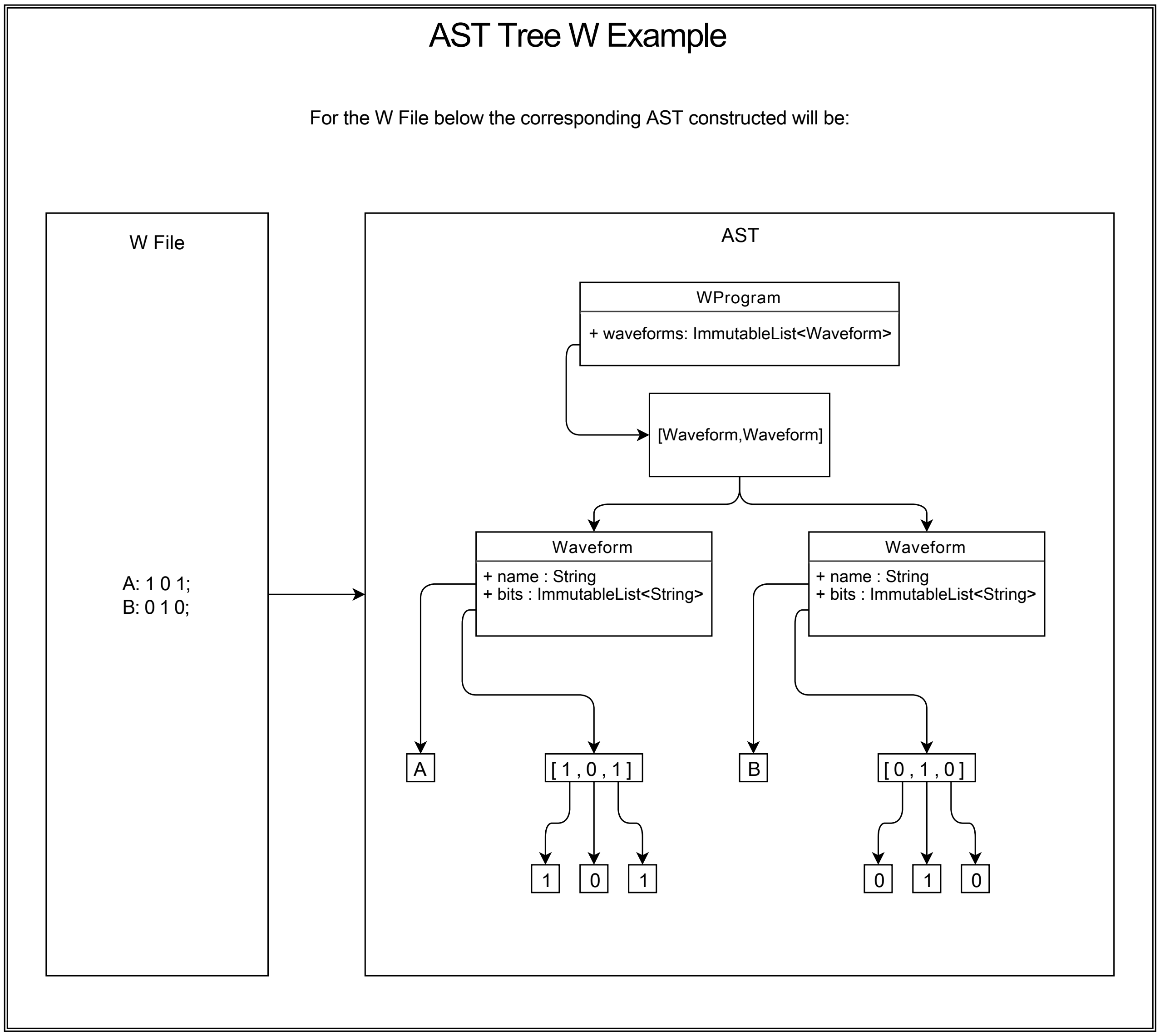

AST Tree W Example
For the W File below the corresponding AST constructed will be:
W File
A: 1 0 1;
B: 0 1 0;
AST
WProgram
+ waveforms: ImmutableList<Waveform>
[Waveform,Waveform]
Waveform
+ name : String
+ bits : ImmutableList<String>
Waveform
+ name : String
+ bits : ImmutableList<String>
A
[1 , 0 , 1]
B
[0 , 1 , 0]
1
0
1
0
1
0

Figure 1.6: Object diagram for example $\mathcal{W}$ AST

1.8 *Steps to success*

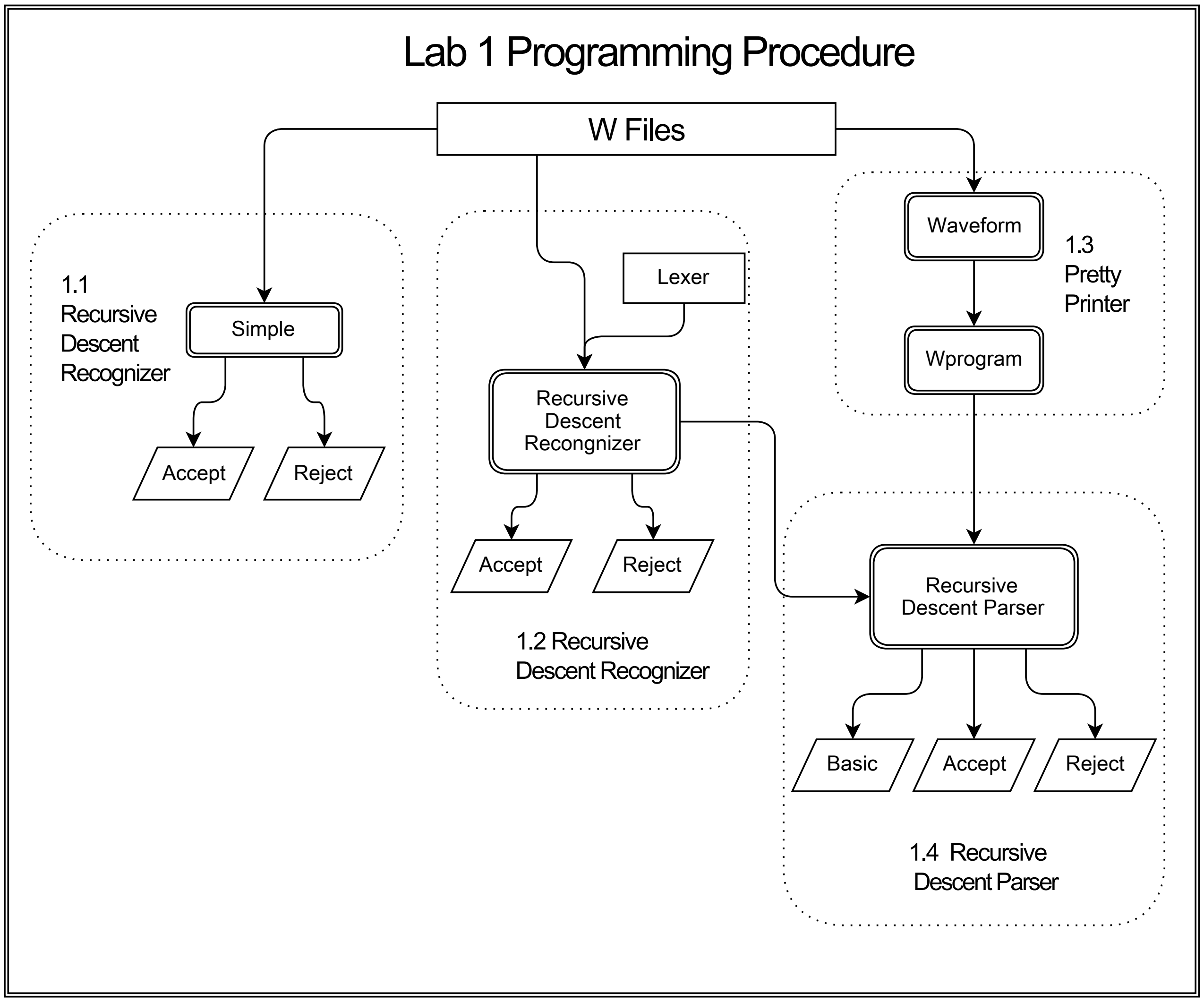
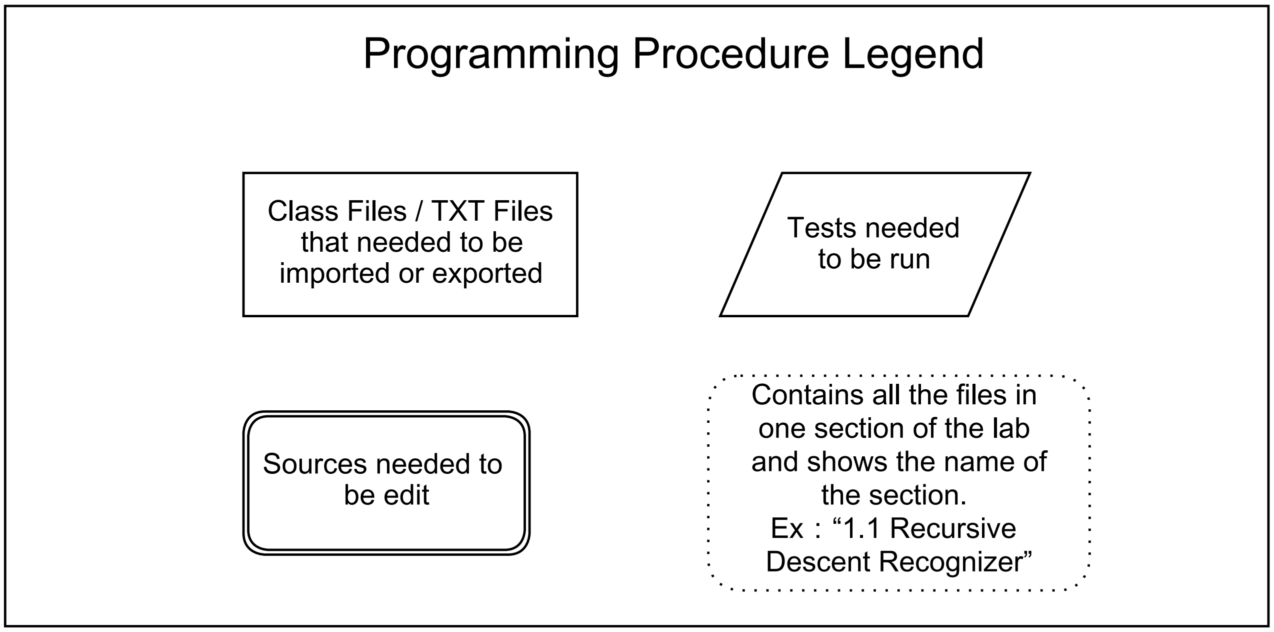

Lab 1 Programming Procedure
W Files
1.1 Recursive Descent Recognizer
Simple
Accept
Reject
Lexer
Recursive Descent Recongnizer
Accept
Reject
1.2 Recursive Descent Recognizer
Waveform
Wprogram
1.3 Pretty Printer
Recursive Descent Parser
Basic
Accept
Reject
1.4 Recursive Descent Parser

Figure 1.7: Steps to success for Lab 1. Legend for icon meanings in Figure 1.8.


Programming Procedure Legend
Class Files / TXT Files that needed to be imported or exported
Tests needed to be run
Sources needed to be edit
Contains all the files in one section of the lab and shows the name of the section. Ex : "1.1 Recursive Descent Recognizer"

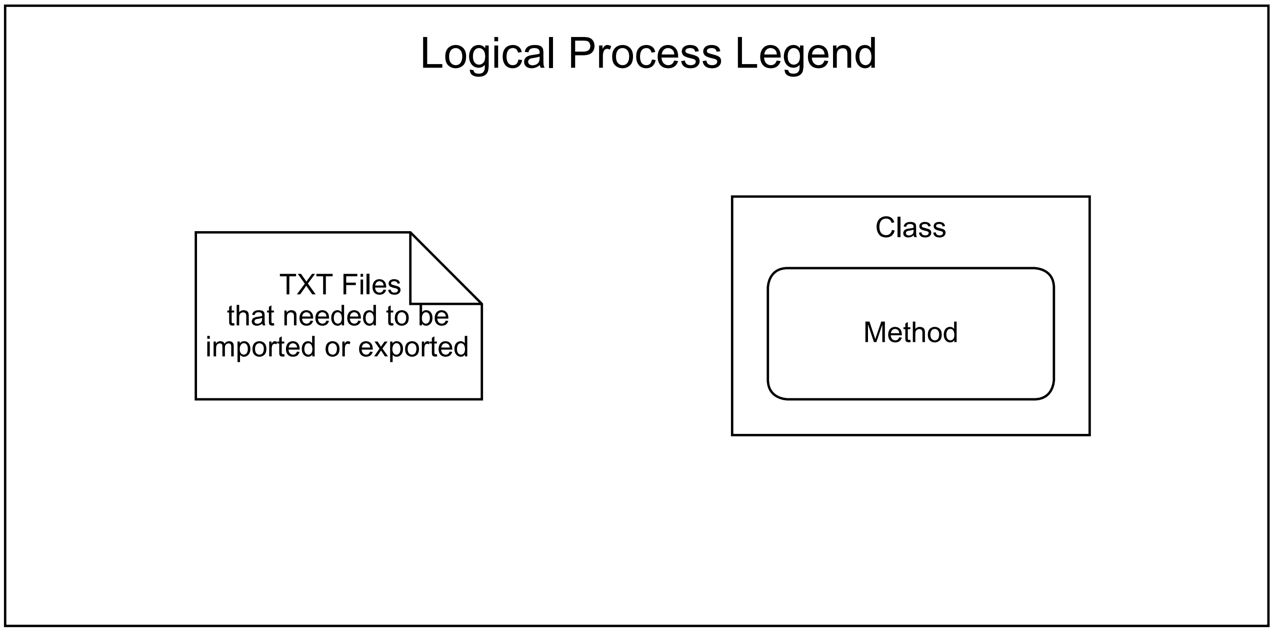


Figure 1.8: Legend for Figure 1.7

1.9 *Evaluation*

The last pushed commit before the deadline is evaluated both on the shared test inputs and on a set of secret test inputs according to the weights in the table below.

Note that you don't earn any points for the rejection tests until some of the corresponding acceptance tests pass. Also, you don't earn any points for the parser until the TestWRDParserBasic tests pass.

	Shared	Secret
TestWRegexAccept	10	5
TestWRegexReject	5	5
TestWRDRecognizerAccept	15	5
TestWRDRecognizerReject	5	5
TestWRDParserBasic	5	0
TestWRDParserAccept	20	10
TestWRDParserReject	5	5

1.10 *Reading*

1.10.1 *Tiger Book*[1]

[1] A. W. Appel and J. Palsberg. *Modern Compiler Implementation in Java*. Cambridge University Press, 2004

- 1 Introduction
- 2.0 Lexical Analysis
- 2.1 Lexical Tokens
- 2.2 Regular Expressions
- 3.0 Parsing
- 3.2.0 Predictive Parsing / Recursive Descent
  - skip First and Follow Sets for now
  - read Constructing a Predictive Parser on p.50
  - skip Eliminating Left Recursion and everything that comes after it — for now
- 4.1.1 Recursive Descent

1.10.2 *Programming Language Pragmatics*[2]

[2] M. L. Scott. *Programming Language Pragmatics*. Morgan Kaufmann, 3 edition, 2009

- 2.2 Scanning
- 2.3.1 Recursive Descent

1.10.3 *Web Resources*

Thinking in Java[3]
Prof Alex Aiken @ Stanford on recursive descent:
http://www.youtube.com/watch?v=Kdx5HOtb-Zo
http://www.youtube.com/watch?v=S7DZdn33eFY

[3] B. Eckel. *Thinking in Java*. Prentice-Hall, 2002. http://www.mindview.net/Books/TIJ/

Compiler Concepts: trees, transformations, XML, SVG
Programming Concepts: object contract, DOM *vs.* SAX parser styles, call-backs, iterator design pattern

Lab 2
Transforming $\mathcal{W} \rightarrow$ SVG for Visualization

While our circuit simulator programs will read and write $\mathcal{W}$ files, people often prefer to look at waveforms in a graphical format. In this lab we will translate $\mathcal{W}$ files into SVG files for visualization.

SVG is a an XML-based graphics file format: in other words, it is structured text that describes some vectors to be rendered. Any modern web browser will render SVG.

An example $\mathcal{W}$ file is shown in Figure 2.1. Figure 2.2 shows what the corresponding SVG looks like when visualized with a web browser such as Firefox or a vector graphics program such as Inkscape. Figure 2.3 shows the first few lines of the textual content of the SVG file.

```
A: 0 1 0 1 0 1 ;
B: 1 0 1 0 1 0 ;
OR: 1 1 1 1 1 1 ;
```

Figure 2.1: Example waveform file for an OR gate. The input pins are named A and B, and the output pin is named OR. Our VHDL simulator will read a $\mathcal{W}$ file with lines A and B and will produce a $\mathcal{W}$ file with all three lines.

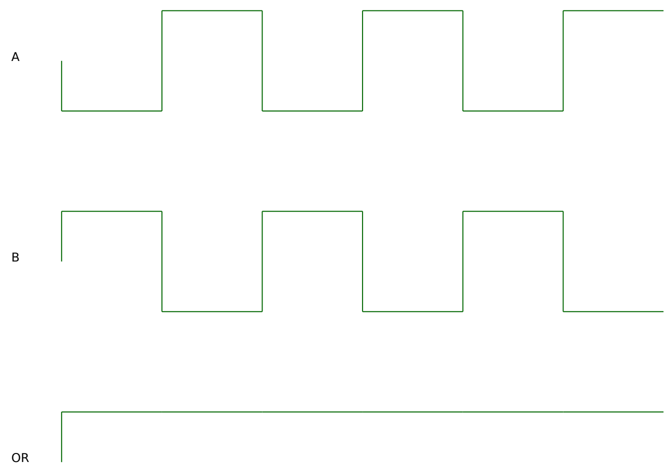


Figure 2.2: Rendered SVG of $\mathcal{W}$ file from Figure 2.1

```
<?xml version="1.0" encoding="UTF−8"?>
<!DOCTYPE svg PUBLIC "−//W3C//DTD SVG 1.1//EN" "http://www.w3.org/
     Graphics/SVG/1.1/DTD/svg11.dtd">
<svg xmlns="http://www.w3.org/2000/svg" width="100%" height="100%"
     version="1.1">
<style type="text/css"><![CDATA[line{stroke:#006600;fill:#00 cc00;} text{font−
     size:"large";font−family:"sans−serif"}]]></style>

<text x="50" y="150">A</text>
<line x1="100" x2="100" y1="150" y2="200" />
<line x1="100" x2="200" y1="200" y2="200" />
<line x1="200" x2="200" y1="200" y2="100" />
<line x1="200" x2="300" y1="100" y2="100" />
```

Figure 2.3: First 10 lines of SVG text of $\mathcal{W}$ file from Figure 2.1. The boilerplate at the top is already in W2SVG.java for your convenience.

Note that the origin of an SVG canvas is the top left corner, not the bottom left corner (as you might expect from math). Consequently, Y=200 is visually lower than Y=100, since Y counts down from the top.

2.1 *Write a translator from* $\mathcal{W} \rightarrow$ SVG

The idea of the transformation is simple: for each bit (zero or one) in the input file, produce a vertical line and a horizontal line. To do this the transformer needs to remember three things: the current X and Y position of the (conceptual) cursor, and the YOffset so we can move the cursor down the page when we start drawing the next waveform.

Run your transformations and inspect the output both visually, in a program such as Firefox, and textually, using the text editor of your choice. Ensure that the output is sensible.

Sources:
ece351.w.svg.TransformW2SVG
Libraries:
ece351.w.ast.WProgram
ece351.w.svg.Line
ece351.w.svg.Pin
Tests:
ece351.w.svg.TestW2SVG

$\mathcal{Q}$: Where is the origin on an SVG canvas? Which directions are positive and which are negative?

2.2 *Write a translator from* SVG $\rightarrow \mathcal{W}$

This translator is the inverse of the previous one. We write this inverse function as a way to test the previous translator. Now we can take advantage of the general property that $x = f'(f(x))$ or, in other words, $w = toW(toSVG(w))$.

This inverse translation is a bit tricky because the SVG files do not contain any explicit information about which line segments are associated with which waveform: the SVG file just contains a bunch of text labels and a bunch of line segments. We have to infer, from the y values, which line segments belong to which waveform/label. Then we use the x values to infer the ordering of the bits, and the y values to infer the bit values (0 or 1).

Many program analysis tasks used in optimizing compilers involve this kind of inference: recovering information that is implicit in a lower-level representation but was explicit in some higher-level representation.

Sources:
ece351.w.svg.TransformSVG2W
Libraries:
ece351.w.ast.WProgram
ece351.w.svg.Line
ece351.w.svg.Pin
Tests:
ece351.w.svg.TestW2SVG2W

2.3 Introducing the Object Contract[†]

Sources:
ece351.objectcontract.TestObjectContract
Libraries:
ece351.objectcontract.TestObjectContractBase
Tests:
ece351.objectcontract.TestObjectContract

The object contract looks simple (and it is), but even expert programmers have difficulty getting it right in all circumstances.

The object contract is a term sometimes used to describe properties that all objects should have. It is particularly concerned with the *equals* and *hashCode* methods. The equals method should represent a *mathematical equivalence relation* and it should be consistent with the hashCode method. A mathematical equivalence class has three properties:

reflexive	x.equals(x)
symmetric	x.equals(y) ⇔ y.equals(x)
transitive	x.equals(y) && y.equals(z) ⇒ x.equals(z)

Additionally, by consistent with hashCode we mean the following:

consistent	x.equals(y) ⇒ x.hashCode() == y.hashCode()

Finally, no object should be equal to null:

not null	!x.equals(null)

Q : What would happen if every hashCode method returned 42?

Q : What would happen if every hashCode method returned a random value each time it was called?

The file TestObjectContract contains some code stubs for testing the object contract. In order to fill in those stubs correctly you might need to read and understand the code in TestObjectContractBase.

What is the default behaviour of the equals and hashCode methods? In other words, what happens if you do not implement those methods for your objects? Consider the code listings in Figure 2.4. What value is printed out by each statement?

The == ('double equals' or 'equals equals') comparison operator compares the memory addresses of the objects referred to by the two variables.

```
Integer x = new Integer(1);
Integer y = new Integer(1);
System.out.println(x.equals(y));
System.out.println(x == y);
```

```
List a = new LinkedList();
List b = new LinkedList();
System.out.println(a.equals(b));
System.out.println(a == b);
```

```
Object p = new Object();
Object q = new Object();
System.out.println(p.equals(q));
System.out.println(p == q);
```

Figure 2.4: Objects with and without implemented equals methods

2.3.1 Learning to write professional code

It is important, in general, for you to learn the Object Contract because it is a set of properties that must hold for all code you write in any object-oriented language (Java, C++, C#, Python, *etc.*). It is also specifically important that you learn the Object Contract to do the labs in this course, because checking the equality of objects forms the basis of the automated marking scripts. In this exercise you are also learning how to write professional code, with the Object Contract as an example. The steps that we are following here are:

a. Identify the mathematical properties that the code should have. (*Object contract: reflexive, symmetric, transitive, etc.*)
b. Write methods that check these mathematical properties on some

input object(s). (*e.g.*, checkEqualsIsReflexive(x)) Call these the *property check methods*.

c. Verify that the property check methods return true when expected by using third party inputs (*e.g.*, java.lang.String and java.lang.Integer).
d. Verify that the property check methods return false when expected by constructing pathological control objects with known deviant behaviour. (*e.g.*, constructAlwaysTrue(), constructAlwaysFalse(), constructToggler(), *etc.*)
e. Use the verified property check methods to assess real code. (*We aren't doing this step in this exercise.*)

In future labs we will use staff-provided property check methods for Object Contract properties. The purpose of this exercise is for you to see what these property check methods look like, both so you understand the Object Contract and so you understand how your future labs will be graded.

2.3.2 *Reading*

http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#equals(java.lang.Object)
http://www.artima.com/lejava/articles/equality.html
http://www.angelikalanger.com/Articles/JavaSolutions/SecretsOfEquals/Equals.html
Implementation of java.util.AbstractList.equals()
Effective Java [1]
§0.14 of this lab manual on Testing

[1] J. Bloch. *Effective Java*. Addison-Wesley, Reading, Mass., 2001

Not yet discussed: inheritance, cyclic object structures

2.4 DOM VS. SAX *parser styles*†

Try toggling the field TransformW2SVG.USE_DOM_XML_PARSER and see how that changes the performance of the test harnesses. One is much faster than the other. Why?

DOM-style parsers are the most common, and what we will always write in ECE351. These parsers explicitly construct an AST, which can then be examined and manipulated.

SAX-style, or *event-driven* parsers, on the other hand, do not explicitly construct the AST: they just allow the user to register call-back functions that are triggered when certain productions in the grammar are parsed. These event-driven parsers often use much less memory because they do not explicitly construct an AST. They also often run faster. But they are not suited to complex tasks.

Why can we use a SAX-style parser in our test harnesses for this lab? For a few reasons: our test harness does not need to manipulate the XML tree; our test harness does not need to do any complex analysis of the XML tree; we have separate AST classes (WProgram and Waveform) that we want to construct from the XML input, and the construction of these AST objects is simple enough to be done with an event-driven parser.

Sources:
ece351.w.svg.TransformW2SVG

The terms DOM and SAX are XML-specific, but the ideas are general.
For example, the BCEL library for manipulating Java bytecodes explicitly constructs an AST, whereas the ASM library for manipulating Java bytecodes is event-driven. If the transformation that you want to do is simple enough to be done with ASM then it will probably execute faster if implemented with that library.

The idea of an event-driven parser might not be clear to you from this brief description. Knowing that such a style of parser exists, its general properties, and the kinds of problems it is suited or not suited for is enough for the exam. You are, however, encouraged to try writing a small event-driven parser with an existing XML or Java bytecode library on your own time to get a deeper understanding of this idea.

The Streaming API for XML (StAX) is a new event-driven parser where the application remains in control, instead of putting the parser in control. http://sjsxp.java.net/

If you are interested to read more, this is a good article:
http://geekexplains.blogspot.ca/2009/04/sax-vs-dom-differences-between-dom-and.html

2.5 *Evaluation*

The last pushed commit before the deadline is evaluated both on the shared test inputs and on a set of secret test inputs according to the weights in the table below.

We provide you with about 30 $\mathcal{W}$ files for these equations.

	Current	*New*	*Equation*
TestW2SVG	20	10	legalSVG(TransformW2SVG(w))
TestW2SVG2W	40	10	SVG2W(W2SVG(w)).equivalent(SVG2W(staff.svg))
TestObjectContract	15	5	see §2.3 above

2.6 *Steps to success*

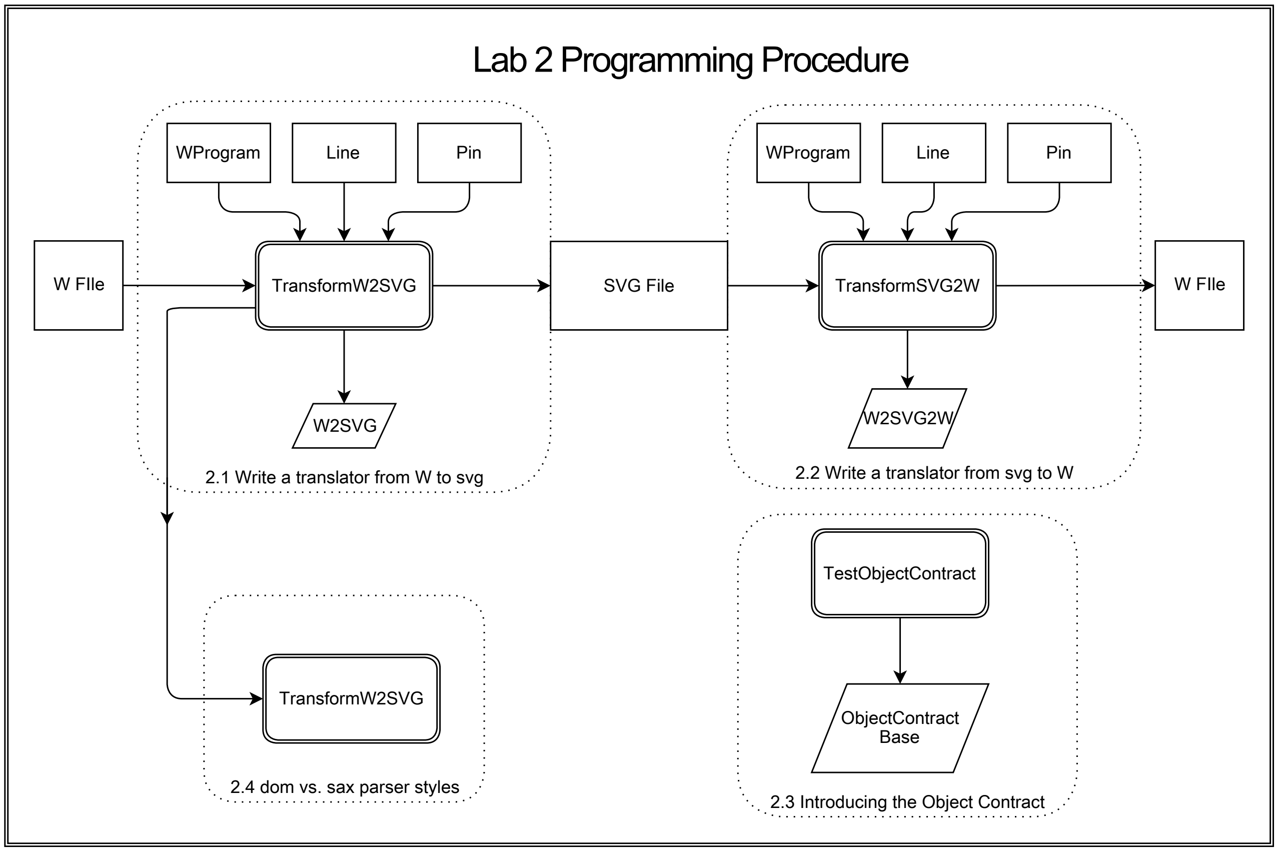

Lab 2 Programming Procedure
WProgram
Line
Pin
W FIle
TransformW2SVG
SVG File
W2SVG
2.1 Write a translator from W to svg
WProgram
Line
Pin
TransformSVG2W
W FIle
W2SVG2W
2.2 Write a translator from svg to W
TestObjectContract
ObjectContract Base
2.3 Introducing the Object Contract
TransformW2SVG
2.4 dom vs. sax parser styles

Figure 2.5: Steps to success for Lab 2. Legend for icon meanings in Figure 1.8.

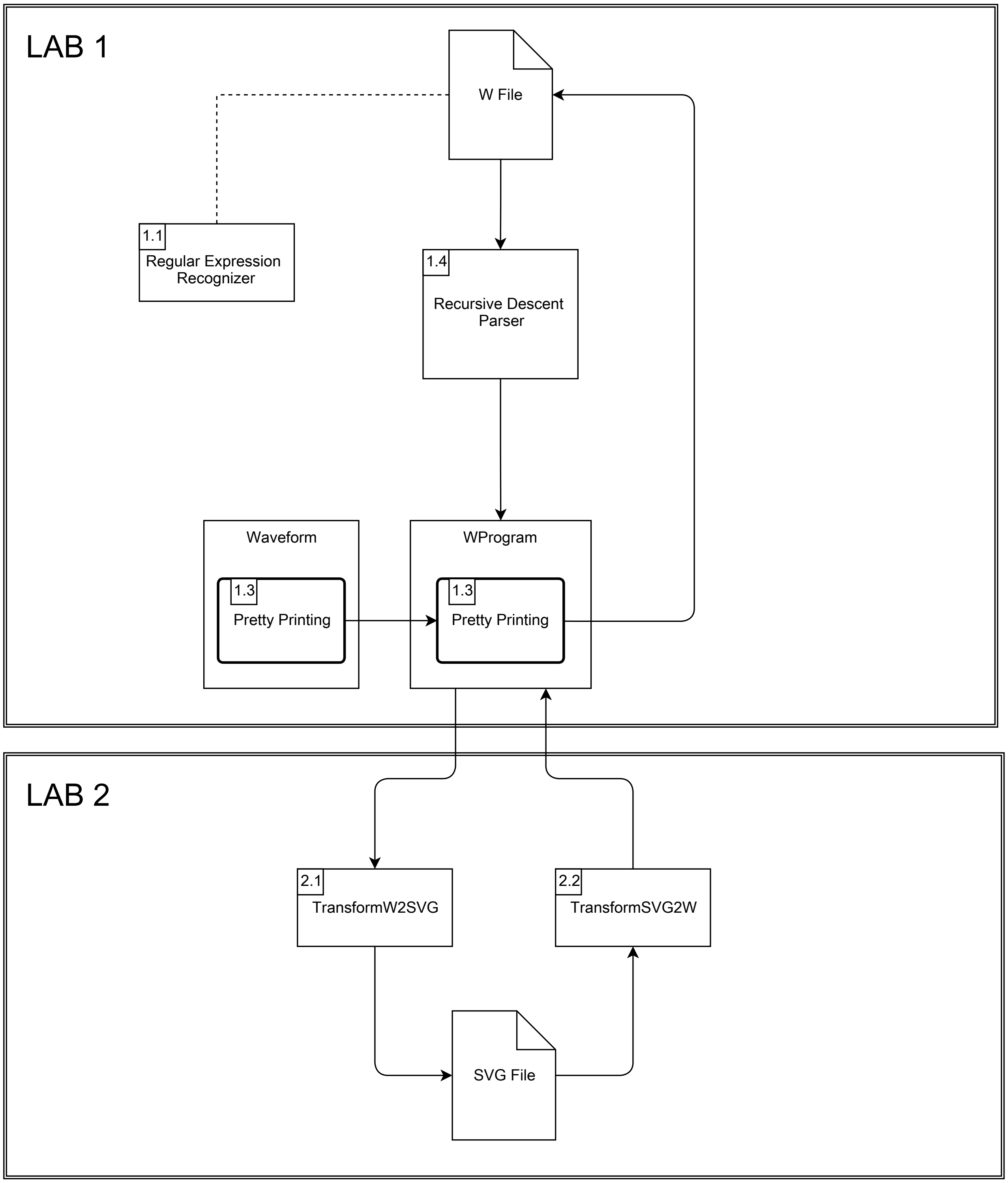


Figure 2.6: How Lab 1 and Lab 2 fit together

Lab 3
Recursive Descent Parsing of $\mathcal{F}$

Compiler Concepts: context-free grammars, LL(1) grammars, predict sets, parse trees, precedence, associativity, commutativity, program equivalence

Programming Concepts: inheritance, polymorphism, dynamic dispatch, type tests, casting, memory safety, composite design pattern, template design pattern, singleton design pattern, recursive functions, recursive structures, higher-order functions

This lab introduces formula language $\mathcal{F}$, which we will use as an intermediate language in our circuit synthesis and simulation tool. In a subsequent lab we will write a translator from VHDL to $\mathcal{F}$.

Compilers usually perform their optimizations on programs in intermediate forms. These intermediate forms are designed to be easier to work with mechanically, at the cost of being less pleasant for people to write large programs in. A program written in $\mathcal{F}$, for example, is just a list of boolean formulae. This is relatively easy to manipulate mechanically. VHDL, by contrast, has conditionals, module structure, *etc.*, which are all of great benefit to the VHDL programmer but are more work for the compiler writer to manipulate. In the next lab we will write a simplifier/optimizer for $\mathcal{F}$ programs.

Program	→	Formula$^+$ $$
Fomula	→	Var '<=' Expr ';'
Expr	→	Term ('`or`' Term)*
Term	→	Factor ('`and`' Factor)*
Factor	→	'`not`' Factor | '(' Expr ')' | Var | Constant
Constant	→	'0' | '1'
Var	→	id

Figure 3.1: LL(1) Grammar for $\mathcal{F}$. $\mathcal{F}$ is a very simple subset of our subset of VHDL, which is in turn a subset of the real VHDL. $\mathcal{F}$ includes only the concurrent assignment statement and the boolean operators conjunction (and), disjunction (or), and negation (not).

3.1 A Tale of Three Hierarchies[†]

There are (at least) three hierarchies involved in understanding this lab — and all of the future labs. In LAB1 we met the *abstract syntax tree* (AST) and the *parse tree* (the execution trace of a recursive descent recognizer/parser). In this lab we will also meet the *class hierarchy* of our Java code. In the past labs there wasn't much interesting in the class hierarchy, but now (and for the remainder of the term) it is a central concern.

3.1.1 *Parse Tree*

Consider the $\mathcal{F}$ program X <= A or B;. When we execute the recursive descent recognizer that we are about to write its call tree will look something like this:

Make sure that you understand this call tree. Try to draw a few on paper for other examples such as:

- X <= A and B;
- X <= A or B and C;
- X <= A and B or C;

These drawings are for you to understand the lab before you start programming. We will not be asking you for the drawings nor will we mark them. If you start programming before understanding it will take you longer to do the lab.

```
program()
    formula()
        var()
            id()
                'X'
        '<='
        expr()
            term()
                factor()
                    var()
                        id()
                            'A'
            'or'
            term()
                factor()
                    var()
                        id()
                            'B'
        ';'
EOI
```

We will use EOI (end of input) and EOF (end of file) and $$ interchangeably.

For the example above and the others mentioned in the margin, also draw the corresponding AST. Both the call tree and the resulting AST depend on the input.

See the readings, especially Bruce Eckel's free online book *Thinking in Java*, for more background material on inheritance/sub-classing.

3.1.2 AST

Recall that 'AST' stands for *abstract syntax tree*, and is the important information that we want to remember from the parse tree above. This important information is the structure of the tree and the interesting nodes. Things like term, factor, and parentheses are ways that the input string communicates its structure to the parser: the AST doesn't need to retain those things, just the structure itself.

```
FProgram
    AssignmentStatement
        outputVar = X
        expr = OrExpr
                left = A
                right = B
```

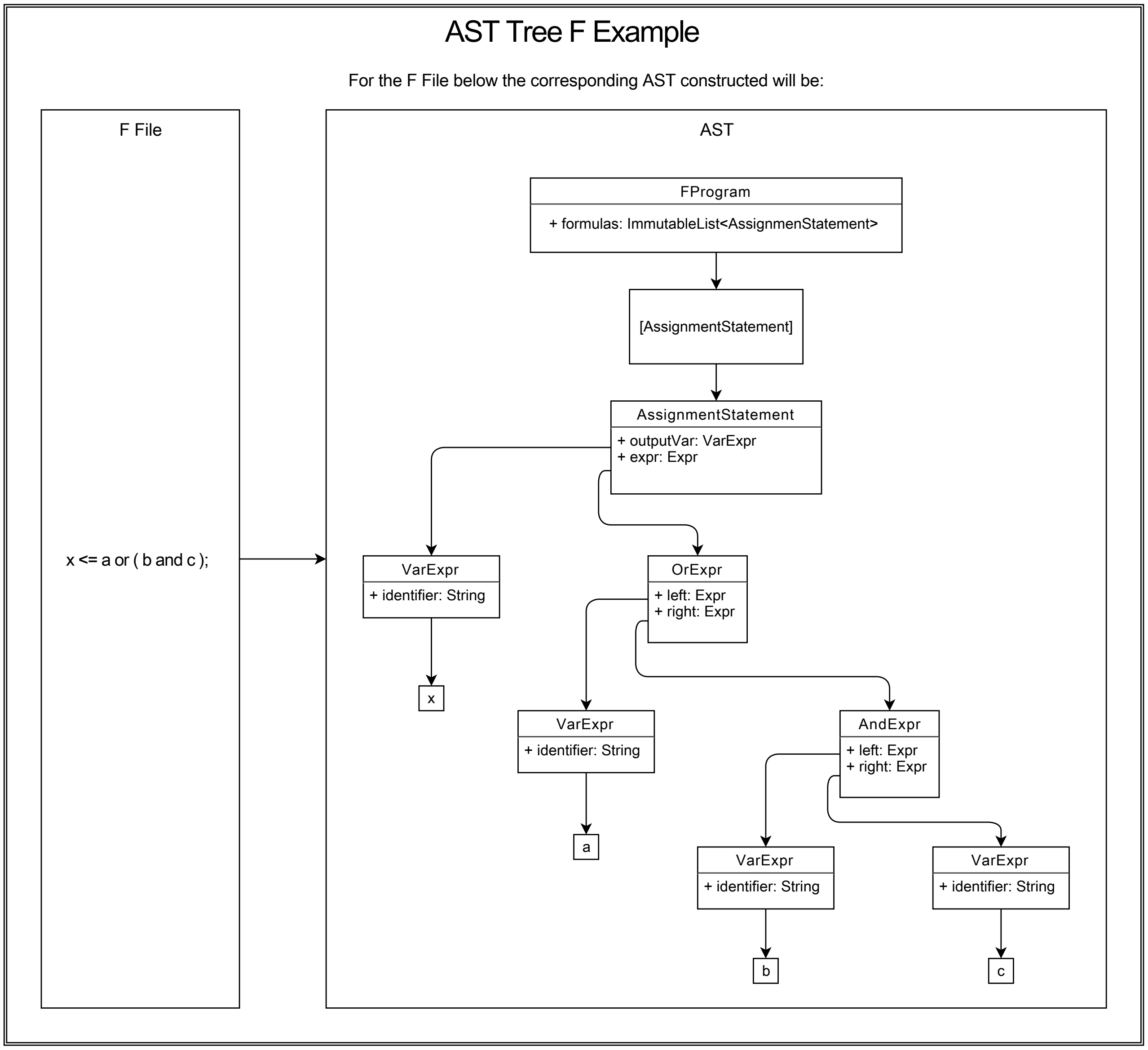


Figure 3.2: Object diagram for example $\mathcal{F}$ AST

3.1.3 *A peak at the Expr class hierarchy*

The class hierarchy, on the other hand, does not depend on the input: it is how the code is organized. Find the class common.ast.Expr in Eclipse, right-click on it and select *Open Type Hierarchy*. This will give you an interactive view of the main class hierarchy. Draw this hierarchy in a UML class diagram, excluding the classes in common.ast that you do not need for this lab. For this lab you will need the classes AndExpr, OrExpr, NotExpr, VarExpr, and ConstantExpr. You will also need the classes that those depend on, but you will not need classes corresponding to more esoteric logical operators such as exclusive-or. This lab also uses classes outside of the Expr class hierarchy, but you don't need to draw them on your UML class diagram.

The following listing highlights some features of the Expr class hierarchy and also demonstrates polymorphism/dynamic-dispatch.

```
abstract class Expr {
    abstract String operator();
}
abstract class BinaryExpr extends Expr {
    final Expr left;
    final Expr right;
    abstract BinaryExpr newBinaryExpr(Expr l, Expr r);
    public String toString() { return left + operator() + right; }
}
final class AndExpr extends BinaryExpr {
    String operator() { return "and"; }
    BinaryExpr newBinaryExpr(Expr l, Expr r) { return new AndExpr(l,r); }
}
final class OrExpr extends BinaryExpr {
    String operator() { return "or"; }
    BinaryExpr newBinaryExpr(Expr l, Expr r) { return new OrExpr(l,r); }
}
final class Main {
    public static void main(String[] args) {
        BinaryExpr e = new AndExpr(); // why isn't the type of e AndExpr?
        Expr[] a = new Expr[3]; // an empty array of size 3
        // what is the type of the object stored in each element of the array?
        a[0] = e;
        a[1] = new OrExpr();
        a[2] = e.newBinaryExpr(null,null); // monomorphic call site
        for (int i = 0; i < a.length; i++) {
            System.out.println(a[i].operator()); // polymorphic call site
            System.out.println(a[i].toString()); // mono or polymorphic?
        }
    }
}
```

Figure 3.3: Some highlights from the Expr class hierarchy.

Draw a UML class diagram for this code.

Draw a diagram showing the relationship between the variables and the objects for the main method. (We have drawn this kind of diagram on the board in class, and these kinds of diagrams are also drawn by PythonTutor.com.)

Annotate this diagram with the *static type* of each variable and the *dynamic type* of each object. In the past we only considered the case where the static type of the variable was the same as the dynamic type of the object referred to by that variable. Now we also consider the case where the dynamic type of the object is a subtype of the variable's static type.

Is there any aliasing occurring in the main method? If so, what is it?

A *monomorphic call site* is one where the dynamic dispatch will always resolve to the same target. A *polymorphic call site* is one where the dynamic dispatch might resolve to a different target each time the call is executed.

The Java compiler/runtime system inserts code like the following at each potentially polymorphic call site:

```
if (a[i] instanceof AndExpr) {
    return AndExpr::operator(); }
else if (a[i] instanceof OrExpr) {
    return OrExpr::operator(); }
```

This feature of implicit type tests to determine which method definition to execute is one of the key features of object-oriented programming. Many design patterns are about ways to organize code with this feature that make certain kinds of anticipated changes to the software modular.

3.2 *Write a recursive-descent recognizer for $\mathcal{F}$*

Figure 3.1 lists a grammar for $\mathcal{F}$. A recognizer merely computes whether a sentence is generated by a grammar: *i.e.*, its output is boolean. A parser, by contrast, also constructs an abstract syntax tree (AST) of the sentence that we can do things with. A recognizer is simpler and we will write one of them first.

The idea is simple: for each production in the grammar we make a function in the recognizer. These functions have no arguments and return void. All these functions do, from a computational standpoint, is examine the lexer's current token and then advance the token. If the recognizer manages to push the lexer to the end of the input without encountering an error then it declares success.

Sources:
ece351.f.rdescent.FRecursiveDescentRecognizer
Libraries:
ece351.util.Lexer
ece351.util.CommandLine
Tests:
ece351.f.rdescent.TestFRDRecognizerAccept
ece351.f.rdescent.TestFRDRecognizerReject
Reflect: Can you write a regular expression recognizer for $\mathcal{F}$? No, because $\mathcal{F}$ allows nested expressions / balanced parentheses. Checking that the parentheses are balanced requires a pushdown automata, which is strictly more powerful the finite state machine that would be adequate to recognize a regular language.

3.3 *Write a pretty-printer for the* AST

Pretty-printing is the inverse operation of parsing: given an AST, produce a string. (Parsing produces an AST from a string.) In this case the task is easy: implement the toString() methods of the AST classes. When the program invokes toString() on the root node of the AST the result should resemble the original input string.

Sources:
ece351.f.ast.*
ece351.common.ast.*
Tests:
manual inspection of output
parser tests below

3.4 *Equals, Isomorphic, and Equivalent*†

We were previously introduced to the *object contract* and the *equals* method, and learned that all of the equals methods together are supposed to define a *partitioning* of the objects, which has three properties: reflexivity, symmetry, and transitivity. But we did not talk about the specific semantics of the equals method. For example, always returning true from every equals method would define a partitioning: there would just be one partition and every object would be in it. That's not particularly useful.

A *partitioning* is also known as a *mathematical equivalence class*. We'll try to move towards the term 'partitioning' and away from 'mathematical equivalence class', because we use the words 'equivalence' and 'class' in other programming contexts.

In this lab we will give some more meaningful semantics to the equals method, and we will also define two other partitionings with different semantics: *isomorphic* and *equivalent*. Here is an example:

$$\text{X <= A or !A;} \overset{equals}{\longleftrightarrow} \text{X <= A or !A;} \overset{isomorphic}{\longleftrightarrow} \text{X <= !A or A;} \overset{equivalent}{\longleftrightarrow} \text{X <= 1;}$$

equals: Two objects are equals if any computation that uses either one will produce identical results.[1] This can only be true if the objects are *immutable* (*i.e.*, the values stored in their fields do not change).

[1] B. Liskov and J. Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design.* Addison-Wesley, Reading, Mass., 2001

isomorphic: We will say that two objects are *isomorphic* if they have the same elements and similar structures. For example, we will consider the expressions X or Y and Y or X to be isomorphic: they are permutations of the same essential structure. Any two objects

equals $\Rightarrow$ isomorphic

that are equals are also isomorphic, but isomorphic objects are not necessarily equals. For example, the expressions X or Y and Y or X are isomorphic but not equals.

equivalent: We will say that two objects are *equivalent* if they have the same meaning, but possibly different structures and possibly different elements. For example, the expression 1 is equivalent to the expression X or !X: they have the same meaning but totally different syntax. Any two objects that are isomorphic are also equivalent, but not necessarily *vice versa* (as in this example).

You will not be implementing equivalent this term, but you will eventually (not for this lab) need to understand what it means and how we have implemented it for FProgram.

isomorphic ⇒ equivalent

3.5 Write Equals and Isomorphic for the $\mathcal{F}$ AST classes

Sources:
ece351.common.ast.VarExpr
ece351.common.ast.ConstantExpr
ece351.common.ast.AssignmentStatement
ece351.common.ast.UnaryExpr
ece351.common.ast.CommutativeBinaryExpr
ece351.f.ast.FProgram
Libraries:
ece351.util.Examinable
ece351.util.Examiner
ece351.util.ExaminableProperties
Tests:
ece351.f.TestObjectContractF

Start with VarExpr and ConstantExpr. You should be able to fill in these skeletons with the knowledge you have learned so far.

For the FProgram and AssignmentStatement classes you will need to make recursive function calls. Understanding why these calls will terminate requires understanding recursive object structures.

For the UnaryExpr and CommutativeBinaryExpr classes you will also need to understand inheritance and polymorphism. Why do we use getClass() for type tests instead of instanceof? Why do we cast to UnaryExpr instead of NotExpr? How does the operator() method work? *etc.*

3.6 Write a recursive-descent parser for $\mathcal{F}$

Sources:
ece351.f.rdescent.FRecursiveDescentParser
Tests:
ece351.f.rdescent.TestFRDParserBasic
ece351.f.rdescent.TestFRDParser

Our recursive-descent parser will follow the same structure as our recursive-descent recognizer. The steps to write the parser are the same as in the previous lab:

- Copy the procedures from the recognizer.
- For each procedure, change its return type from void to one of the AST classes.
- Modify each procedure to construct the appropriate AST object and return it.

It is important that you follow these steps. If you try to implement the parser for $\mathcal{F}$ in one big method you will run into a lot of trouble.

Reflect: Is the result of pretty-printing an AST always character-by-character identical with the original input program? Write a program in $\mathcal{F}$ for which the result of pretty-printing the AST is not identical to the original input string.

3.7 *Steps to success*

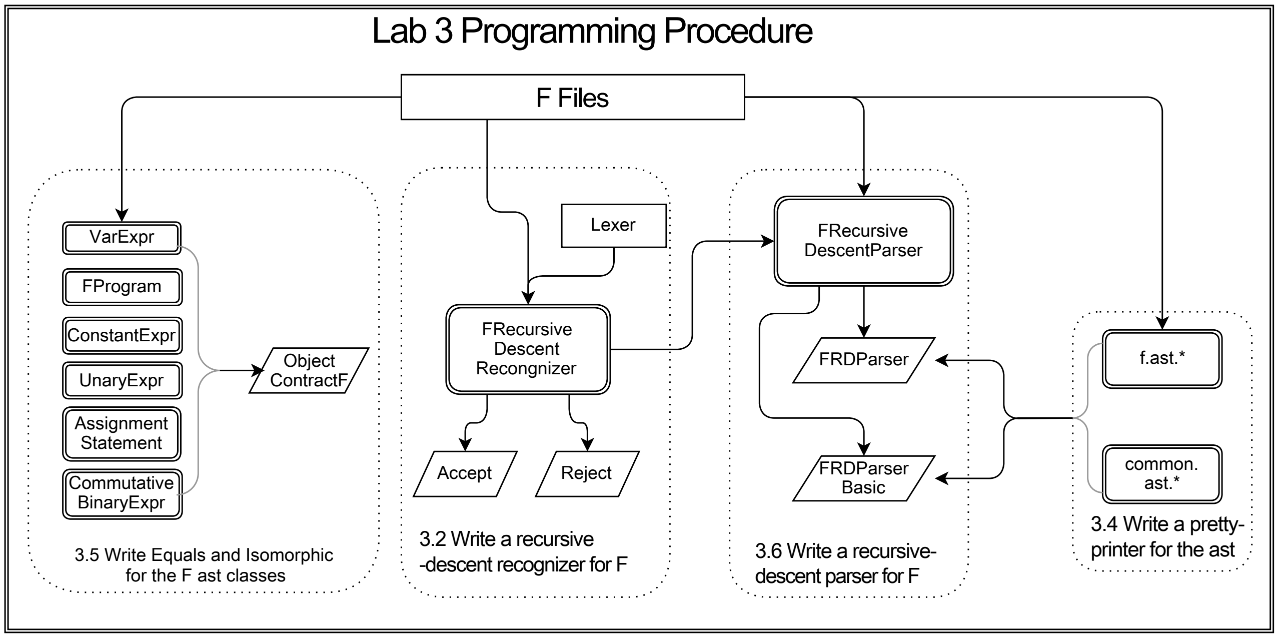

Lab 3 Programming Procedure
F Files
VarExpr
FProgram
ConstantExpr
UnaryExpr
Assignment Statement
Commutative BinaryExpr
Object ContractF
3.5 Write Equals and Isomorphic for the F ast classes
Lexer
FRecursive Descent Recongnizer
Accept
Reject
3.2 Write a recursive -descent recognizer for F
FRecursive DescentParser
FRDParser
FRDParser Basic
3.6 Write a recursive-descent parser for F
f.ast.*
common. ast.*
3.4 Write a pretty-printer for the ast

Figure 3.4: Steps to success for Lab 3. Legend for icon meanings in Figure 1.8.

3.8 *Evaluation*

The last pushed commit before the deadline is evaluated both on the shared test inputs and on a set of secret test inputs according to the weights in the table below.

The testing equation for this lab is:

$\forall$ AST | AST.equals(parse(prettyprint(AST)))

At present we have about 80 AST's to plug into this equation. We might explore some mechanical techniques to generate more AST's.

One of the great benefits of having a testing equation is that we can separate out the correctness condition (the equation) from the inputs. Then we can look for other techniques to generate inputs. By contrast, if we test with just specific input/output pairs the correctness condition is that we get the output for that input.

	Current	New
TestFRDRecognizerAccept	5	5
TestFRDRecognizerReject	5	5
TestFRDParserBasic	5	5
TestFRDParser	30	10
TestObjectContractF	20	10

3.9 *Background & Reading*

See also the readings for LAB1 above.

Tiger Book

3.0 Parsing

3.1 Context-Free Grammars

3.2.0 Predictive Parsing / Recursive Descent

- including First and Follow Sets
- read Constructing a Predictive Parser on p.50
- skip Eliminating Left Recursion and everything that comes after it — for now (we'll study this for exam, but not for the labs)

4.1.1 Recursive Descent

Thinking in Java [2] is a good resource for object-oriented programming in general and Java in particular.

[2] B. Eckel. *Thinking in Java*. Prentice-Hall, 2002. http://www.mindview.net/Books/TIJ/

1 Introduction to Objects
6 Reusing Classes
7 Polymorphism

Some topics you should be comfortable with include:

If you are not already familiar with these topics then this lab is going to take you longer than five hours. Once you get through these background topics then the rest of the course material should be fairly straightforward.

- inheritance / subtyping / subclassing
- polymorphism / dynamic dispatch
- objects *vs.* variables, dynamic *vs.* static type
- type tests, casting

ECE155 Lecture Notes In winter 2013 ECE155 switched to Java (from C#), and the third lecture covered some of the differences.

http://patricklam.ca/ece155/lectures/pdf/L03.pdf

Lab 4
Circuit Optimization: $\mathcal{F}$ Simplifier

Compiler Concepts: intermediate languages, identity element, absorbing element, equivalence of logical formulas, term rewriting, termination, confluence, convergence
Programming Concepts: interpreter design pattern, template design pattern, representation invariants

$\mathcal{F}$ is an intermediate language for our circuit synthesis and simulation tools, in between VHDL and the final output languages, as was depicted in Figure 1. In a subsequent lab we will write a translator from VHDL to $\mathcal{F}$.

Compilers usually perform their optimizations on programs in an intermediate language. These intermediate languages are designed to be easier to work with mechanically, at the cost of being less pleasant for people to write large programs in. A program written in $\mathcal{F}$, for example, is just a list of boolean formulas. This is relatively easy to manipulate mechanically. VHDL, by contrast, has conditionals, module structure, *etc.*, which are all of great benefit to the VHDL programmer but are more work for the compiler writer to manipulate.

The simplifier we will develop in this lab will work by *term-rewriting*. For example, when it sees a term of the form $x + 1$ it will rewrite it to 1. Figure 4.1 lists the algebraic identities that your simplifier should use.

Our simplifier works at a syntactic level: *i.e.*, it does not have a deep understanding of the formulas that it is manipulating. You may have previously studied semantic techniques for boolean circuit simplification such as Karnaugh Maps and the Quine-McCluskey algorithm for computing prime implicants.

By completing this lab you will have the necessary compiler-related knowledge to implement these more sophisticated optimization algorithms if you want to.

4.1 Simplification Rules

Figure 4.1 gives an algebraic summary of the simplification rules that we will use in this lab, and which files those rules will be implemented in. Before we get to actually implementing these rules there is another transformation that we need to perform, described next.

Level	Description	Code
1	Constant Folding	NotExpr, NaryExpr
		$!0 \mapsto 1$ $!1 \mapsto 0$ $x \cdot 0 \mapsto 0$ $x \cdot 1 \mapsto x$ $x + 0 \mapsto x$ $x + 1 \mapsto 1$
2	Complement	NotExpr, NaryExpr
		$!!x \mapsto x$ $x \cdot !x \mapsto 0$ $x + !x \mapsto 1$
3	Deduplication	NaryExpr
		$x + x \mapsto x$ $x \cdot x \mapsto x$
4	Absorption	NaryExpr
		$x + (x \cdot y) \mapsto x$ $x \cdot (x + y) \mapsto x$ $(a + b) + ((a + b) \cdot y) \mapsto a + b$

Figure 4.1: Optimizations for $\mathcal{F}$ programs, stratified into levels and described by algebraic identities where possible.

There are two possible interpretations of the absorption identity given in the table: First, that x is a VarExpr. Second, that x is an NaryExpr. The latter case is harder.

4.2 *Transforming BinaryExprs to NaryExprs*

To test your simplifier we'll need to compare the output it computes with the output that the staff simplifier computes. In a previous lab we wrote an *isomorphic* method that accounted for the commutativity property of many binary operations: *e.g.*, it would consider $x + y$ and $y + x$ to be isomorphic. That's a good start, but we'll need more than this to really evaluate your simplifier mechanically.

Where to implement this idea will be described below.

We say that an operator is *binary* because it has two operands. It is just a coincidence that our binary operators operate on boolean values. For example, arithmetic addition is also a binary operator, even though it operates on numbers. We say that an operator is *n-ary* if it has a variable number of operands, possibly more than two.

Many of the binary operations we consider are also *associative*. That is, $x + (y + z)$ is equivalent to $(x + y) + z$. Disjunction and conjunction are both associative, as are addition and multiplication.

conjunction means 'logical and'
disjunction means 'logical or'

Detecting the equivalence of $x+(y+z)$ and $(x+y)+z$ is tricky because these parse trees are not structurally similar (recall that 'structurally similar' is what *isomorphic* means). We could try to implement a really clever *equivalent* method that detects this equivalence, or we could go all out and compute the truth tables for each expression and compare them, or we could try something else: transforming these trees into a standardized form for which it is easy to check isomorphism.

Figure 4.2 shows four different trees that all represent the same logical expression. The first three trees are binary. Comparing them for equivalence is difficult. However, all three binary trees can be transformed into the sorted n-ary tree fairly easily, and these sorted n-ary trees can be easily compared to each other for isomorphism.

Note that this transformation requires that the operator (*e.g.*, '+') be both associative and commutative. Fortunately, both conjunction and disjunction are associative and commutative, and these are the only operators that we are applying this transformation to.

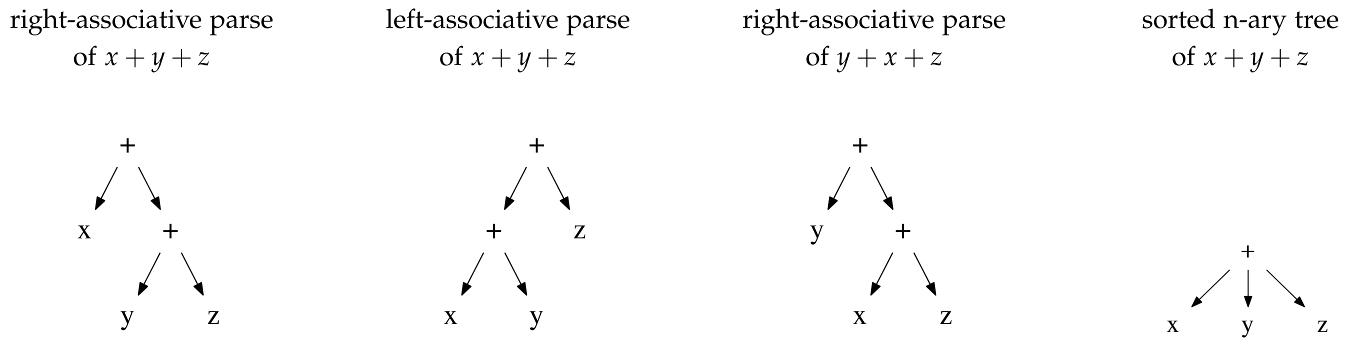

right-associative parse of $x+y+z$
left-associative parse of $x+y+z$
right-associative parse of $y+x+z$
sorted n-ary tree of $x+y+z$
+
x
y
z

Figure 4.2: Four trees that represent logically equivalent circuits. The sorted n-ary representation makes equivalence comparisons easier.

Where to implement this transformation will be described below.

4.3 *Simplify Once*

Sources:
ece351.common.ast.NotExpr
ece351.common.ast.NaryExpr
....NaryOrExpr.getIdentityElement()
....NaryOrExpr.getAbsorbingElement()
....NaryAndExpr.getIdentityElement()
....NaryAndExpr.getAbsorbingElement()
Libraries:
NaryExpr.filter()
NaryExpr.removeAll()
NaryExpr.contains()
NaryExpr.getThatClass()
Tests:
TestSimplifier2

As described above, the work for you in this lab is concentrated in the simplifyOnce() methods in the NotExpr and NaryExpr classes.

Figures 4.3 and 4.4 list the provided implementations of simplifyOnce() for the AndExpr and OrExpr classes, respectively. As can be seen, all these methods do is convert AndExpr/OrExpr objects into NaryAndExpr or NaryOrExpr objects. This code does not do the interesting work described above in section 4.2 though, because these new NaryExpr objects just have two children.

Figure 4.3: AndExpr.simplifyOnce()

```
@Override
protected Expr simplifyOnce() {
        return new NaryAndExpr(left, right);
```

Figure 4.4: OrExpr.simplifyOnce()

```
@Override
public Expr simplifyOnce() {
        return new NaryOrExpr(left, right);
```

You will implement the interesting work of transforming binary expressions into n-ary expressions in the mergeGrandchildren() method of the NaryExpr class. Figure 4.5 shows where this merging method is called by NaryExpr.simplifyOnce() (which is provided for you).

Figure 4.5: NaryExpr.simplifyOnce() has been broken down into multiple steps for you.

You are not obliged to use exactly these steps, but we recommend that you at least follow the type signatures of these methods. Each of these helper methods returns an NaryExpr, except for singletonify(), which returns an Expr.

One way that you can work with these methods is to replace the exception throws with return this;. Then the methods will all compile and run while you work on filling in their bodies.

```
                        removeIdentityElements().
                        removeDuplicates().
                        simpleAbsorption().
                        subsetAbsorption().
                        singletonify();
        assert result.repOk();
        return result;
}


private NaryExpr simplifyChildren() {
        final ImmutableList<Expr> emptyList = ImmutableList.of();
        NaryExpr result = newNaryExpr(emptyList);
```

4.4 *Object sharing in the simplifier*

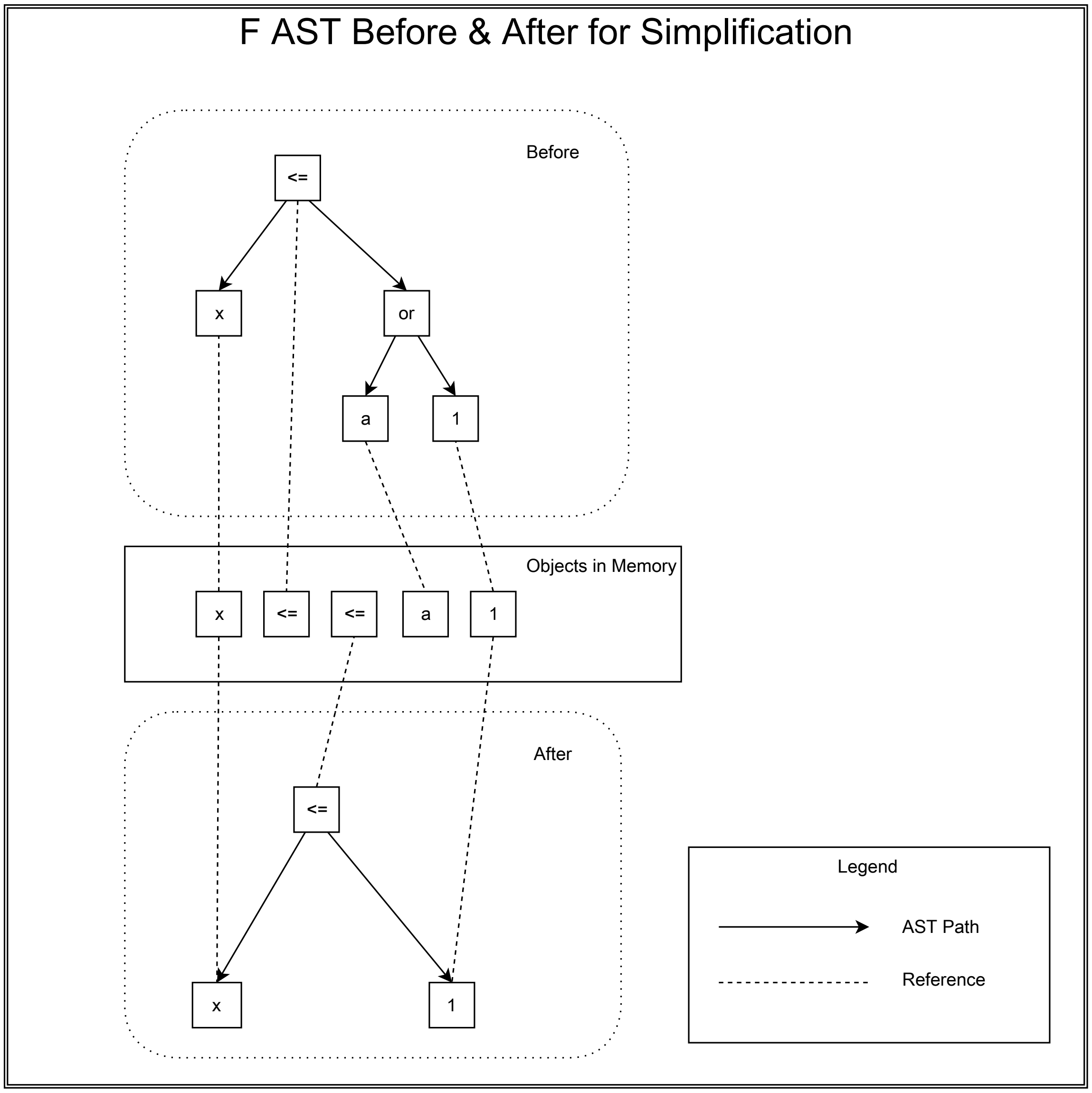


Figure 4.6: Object sharing in the $\mathcal{F}$ simplifier. The original AST and the simplified AST will share some nodes. This is permissible because all of our AST nodes are immutable.

4.5 The Template Method Design Pattern†

A *Template Method* defines an algorithm in outline, but leaves some primitive operations to be defined by the particular datatypes the algorithm will operate on. For example, most sorting algorithms depend on a comparison operation that would be defined differently for strings or integers.

http://en.wikipedia.org/wiki/Template_method_pattern

http://sourcemaking.com/design_patterns/template_method

http://www.oodesign.com/template-method-pattern.html

Typically the template algorithm is implemented in an abstract class that declares abstract method signatures for the primitive operations. The concrete subclasses of that abstract class provide definitions of the primitive operations for their respective data values.

The abstract class does not know the name of its subclasses, nor does it perform any explicit type tests.[1] The type tests are implicit in the dynamic dispatch performed when the primitive operations are called.

[1] An explicit type tests is performed by the instanceof keyword or the getClass() method.

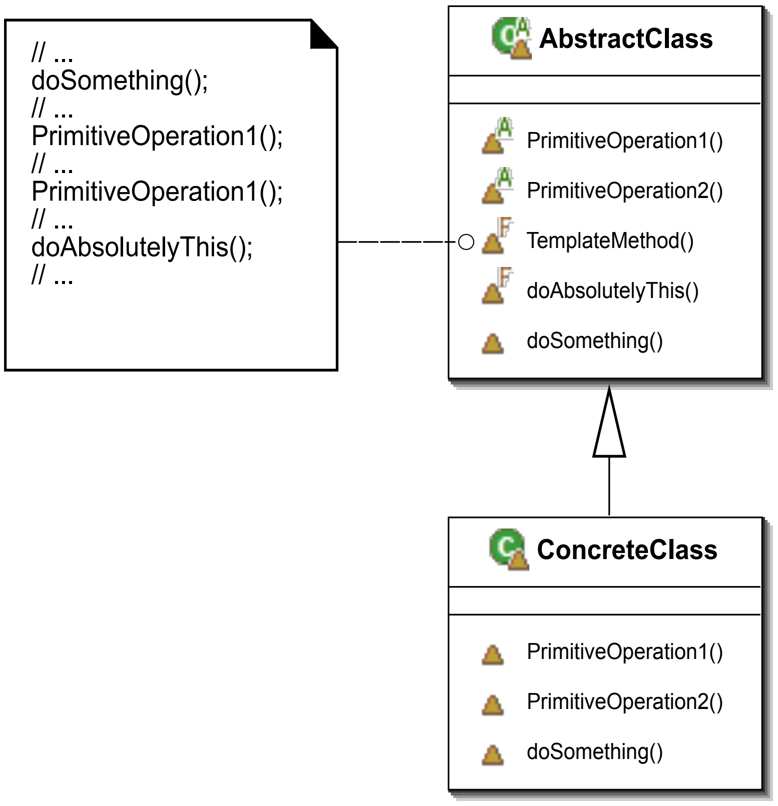


Figure 4.7: Template Design Pattern illustrated by UML Class Diagram. Image from http://en.wikipedia.org/wiki/File:Template_Method_UML.svg. Licensed under GNU Free Documentation Licence.

Expr:

Primitive Op	*Implemented By*	*Template Method*
operator()	every subclass of Expr	toString()
simplifyOnce()	NaryExpr, AndExpr, OrExpr, NotExpr	simplify()

NaryExpr:

Primitive Op	*Implemented By*	*Template Method*
getAbsorbingElement()	NaryAndExpr, NaryOrExpr	simplifyOnce() helpers
getIdentityElement()	NaryAndExpr, NaryOrExpr	simplifyOnce() helpers
getThatClass()	NaryAndExpr, NaryOrExpr	simplifyOnce() helpers

Why?

- No superclass (*e.g.*, NaryExpr) should know what its subclasses (*e.g.*, NaryAndExpr, NaryOrExpr) are.
- Should be able to add a new subclass without perturbing parent (*e.g.*, NaryExpr) and siblings (*e.g.*, NaryAndExpr, NaryOrExpr).
- Explicit type tests (*e.g.*, instanceof, getClass()) usually make for fragile code — unless comparing against own type for purpose of some equality-like operation.
- Use dynamic (polymorphic) dispatch to perform type tests in a modular manner.
- Elegant and general implementation of overall transformations.
- Prevent subclasses from making radical departures from general algorithm.
- Reduce code duplication.

4.6 *Class Representation Invariants*†

http://en.wikipedia.org/wiki/Class_invariant

The *invariants* of a class are conditions that are expected to hold for all legal objects of that class. For example, a legal FProgram will contain at least one formula and will have exactly one formula for each output pin (*i.e.*, no output pin will be computed by two different formulas). A legal NaryExpr will have more than one child, its children must not be null, and its children must be sorted.

By convention, the class representation invariants are checked in a method called repOk().

For mutable objects, invariants are usually expected to hold at all public method boundaries (before the method starts and finishes). All of our AST objects are immutable. We take the position that it is possible to construct illegal AST objects (where the invariants do not hold) as temporary values. Consider the code for NaryExpr.simplifyOnce() in Figure 4.5: each of the helper methods (except singlotonify) is allowed to construct and return an illegal NaryExpr, but at the end we expect to have a legal Expr object.

4.7 *Mathematical Properties of Binary Operators*†

These are things you've probably learned and forgotten many times. Turns out they are actually important in this course.

The term *binary operators* refers to the grammatical fact that these operators take two operands (arguments) —- not to the semantic issue that some operators apply to boolean (binary) values. For example, conjunction (logical and) is binary because it takes two operands, not because it operates on true/false values.

4.7.1 *Commutativity*

Changing the order of the arguments doesn't matter, *e.g.*:

$$x + y = y + x$$

Addition, multiplication, conjunction (and), disjunction (or) are all commutative. Subtraction and division are not commutative: the order of the operands (arguments) matters.

4.7.2 *Associativity*

Wikipedia says it nicely:[2]

> Within an expression containing two or more occurrences in a row of the same associative operator, the order in which the operations are performed does not matter as long as the sequence of the operands is not changed. That is, rearranging the parentheses in such an expression will not change its value. Consider, for instance, the following equations:

[2] http://en.wikipedia.org/wiki/Associative_property

$$1 + (2 + 3) = (1 + 2) + 3$$

Addition, multiplication, conjunction (logical and), disjunction (logical or) are all associative.

Subtraction, division, exponentiation, and vector cross-product are not associative. The terminology can be a little bit confusing here. On the one hand we say that subtraction is *not associative*, and on the other hand we say it is *left associative* because:

$$x - y - z = (x - y) - z$$

Exponentiation is *right associative*:

$$x^{y^z} = x^{(y^z)}$$

Saying that an operator is *not associative* means that it is either *left associative* or *right associative*.

4.8 Identity Element & Absorbing Element†

Let I represent the *identity element* of a binary operation $\otimes$ on elements of set S, and let A represent the *absorbing element* of that operation on that set. Then the following equations hold:

$$\begin{array}{ll} \forall\, x \in S \mid & x = x \otimes I \\ \forall\, x \in S \mid & A = x \otimes A \end{array}$$

What are the absorbing and identity elements for conjunction and disjunction in boolean logic?

For example, with integer addition and multiplication we have:

$$\begin{array}{ll} \forall\, x \in \mathbb{Z} \mid & x = x \times 1 \\ \forall\, x \in \mathbb{Z} \mid & 0 = x \times 0 \\ \forall\, x \in \mathbb{Z} \mid & x = x + 0 \end{array}$$

4.9 Iteration to a Fixed Point†

A common technique in optimizing compilers is to *iterate to a fixed point*: that is, to keep applying the optimizations until the program being transformed stops changing. Figure 4.8 shows the code in Expr.simplify() that we will run in this lab to iterate to fixed point.

We will use this idea of iterating to a fixed point on paper in the course notes when analyzing a grammar to determine if it is LL(1) and when doing dataflow analysis.

```
        Expr e = this;
        while (true) { // loop forever?
                final Expr simplified = e.simplifyOnce();
                if (simplified.equals(e)) {
                        // we're done: nothing changed
                        return simplified;
                } else {
                        // something changed: keep working
                        e = simplified;
                }
        }
}
```

Figure 4.8: Code listing for Expr.simplify() showing iteration to a fixed point. This is the only implementation of simplify() in the project, and it is provided for you. You will be working on implementing simplifyOnce() for some of the AST classes.

Notice the loop while (true). Is this an infinite loop? Will this code ever terminate? Maybe. Inside the body of the loop there is a return statement that will exit the loop (and the method) when simplified.equals(e). We refer to this test to decide whether to exit the loop as the *termination condition*. Will this termination condition ever be true? Will it become true for any possible $\mathcal{F}$ program that we might optimize?

Yes, this termination condition will become true for any possible $\mathcal{F}$ program that we optimize (notwithstanding bugs in your code). How do we know this? Because our optimizer applies rewrite rules that make the formula smaller, and there is a limit to how small a formula can get. Therefore, we say that our term rewriting system is *terminating*: there is no input $\mathcal{F}$ program on which it will get stuck in an infinite loop.

From this termination condition we can reason that the simplify method is *idempotent*: that is, if we apply it twice we get the same result as if we apply it once. We first saw this property above in §0.14 in the form $f(x) = f(f(x))$. In our code here, x.simplify().equals(x.simplify().simplify())

Idempotence is an important general property that can be exploited in testing. It is one of the main general properties that must hold for data synchronizers: if a synchronization is performed, and no data changes on either side, then a second synchronization is performed, the second synchronization should not need to perturb the data. http://en.wikipedia.org/wiki/Idempotence

4.10 *Confluence*†

We now know that our term rewriting system is *terminating*: it will never get stuck in an infinite loop. But will it always compute the same result? What if we apply the rewrite rules in a different order? A term rewriting system that always reaches a unique result for any given input, regardless of the order in which the rules are applied, is called *confluent*.

Suppose, as a counter-example, that we consider a term rewriting system for the square root operator ($\surd$) that has two rewrite rules, one that gives the positive root and another that gives the negative root. This rewrite system is not confluent. For example, $\sqrt{4} \mapsto 2$ and $\sqrt{4} \mapsto -2$, and $2 \neq -2$. It is harder to test code that implements a non-confluent term rewriting system because there could be different outputs for the same input.

The term rewriting system that we are implementing in this lab is confluent. Consider the example in Figure 4.9. Whichever order we apply the rules in we eventually get to the same result.

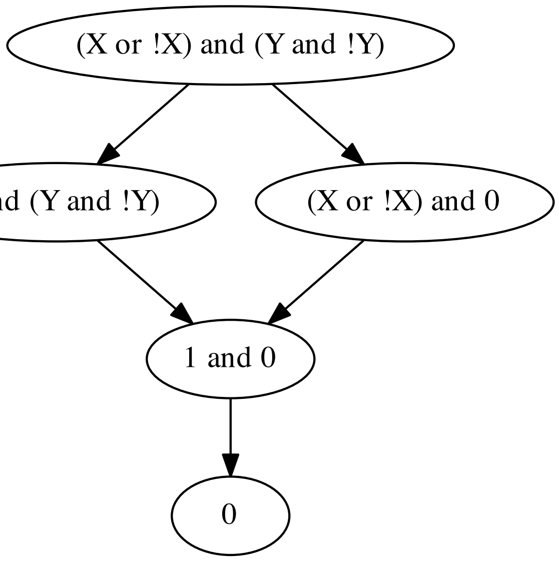


Figure 4.9: Example of two confluent rewrites converging on a common solution

Proving that a term rewriting system is confluent is, in the general case, a fairly difficult problem. For our purposes, we will consider a rewrite system to be confluent if there are no two rules that have the same left-hand side. Our counter-example above had two rules with

convergent = confluent + terminating

$\sqrt{x}$ as the left-hand side. The rules that we are implementing for this lab all have different left-hand sides.

4.11 *Logical equivalence for boolean formulas*†

$\mathcal{F}$ is a language of boolean formulas. Checking equivalence of boolean formulas is an NP-complete problem.

To see why checking equivalence of boolean formulas is NP-complete consider the example of comparing $(x + y) + z$ and $x + (y + z)$ and $!(!x \cdot !y \cdot !z)$ by constructing their truth tables, where f names the output:

$(x$	$+ y)$	$+ z$	$= f$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

$x +$	$(y$	$+ z)$	$= f$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

$!(!x$	$\cdot\, !y$	$\cdot\, !z)$	$= f$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Figure 4.10: Equivalent truth tables

We can see from examining the three truth tables that these three formulas are equivalent. Great. But the number of rows in each truth table is an exponential function of the number of variables in the formula. For example, these formulas have three variables and eight rows in their truth tables: $2^3 = 8$.

What to do? This is a hard problem. As discussed in the *Course Notes* §0 we have three main options: implement an NP-complete algorithm ourselves; use a polynomial time approximation; or convert the problem to SAT and ask a SAT-solver for the answer. If you look at the skeleton code in FProgram.equivalent() you will see that it uses this last approach, and this is the approach that we generally recommend you follow in your future career (unless there are well known and accepted polynomial time approximations for the problem you are trying to solve).

Our translation to SAT occurs via an intermediate language named *Alloy* (http://alloy.mit.edu), which is in turn translated to SAT. We found it convenient to work this way, but it would also be easy to translate this problem to SAT directly.

The SAT solver that Alloy is configured to use in this case is SAT4J, which is open-source, written in Java, and used by Eclipse to compute plugin dependencies.

For this particular problem of computing the equivalence of boolean formulas there is a fourth option: translate the formulas into a standardized form, and then compare that standardized form. This is the approach that we took above when we converted the binary expressions to sorted n-ary expressions. In the case of boolean formulas, the most common standardized form is *reduced, ordered binary decision diagrams* (or BDD for short).[3] A BDD is essentially a compressed form of a truth table.

[3] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, Aug. 1986

Two equivalent boolean formulas will have the exact same BDD representation. Constructing a BDD for a boolean formula is, in the worst case, an exponential problem, but in practice usually runs fairly quickly. BDDs are commonly used for hardware verification and other tasks that require working with boolean formulas.

There will be no exam questions on BDDs. This material is just here for your interest. Boolean formulas are of fundamental and intrinsic interest both in theory and in practice.

4.12 Steps to Success

a. Replace all instances of throw new ece351.util.Todo351Exception(); by return this;[4]

b. Write the examine method for NaryExpr

c. Write the mergeGrandchildern method for NaryExpr

d. Run TestSimplifier2: all of the test cases beginning with opt0 should pass (except for opt0_not_or)

e. Now look at section 4.1 of the lab manual. If you implement all of the simplifications at level 1, all of the opt1 tests should pass.

f. Keep doing step 5 for the next level until you're done.

[4] this step is mentioned above in §4.3

If at step 4 you don't pass any test cases your problem is in one of two methods. Doing it this way will narrow down your debugging and help you get started faster.

4.13 Evaluation

The last pushed commit before the deadline is evaluated both on the shared test inputs and on a set of secret test inputs according to the weights in the table below. The testing equations are as follows. First, the simplifier should always produce an equivalent $\mathcal{F}$ program, for all inputs:

originalAST.simplify().equivalent(originalAST)

Second, the simplifier should be idempotent:

originalAST.simplify().equals(originalAST.simplify().simplify())

Finally, for a select set of $\mathcal{F}$ programs (opt*.f), the simplifier should perform some specific transformations:

originalAST.simplify().isomorphic(staffSimplifiedAST)

In the unlikely, but not impossible, event that you implement a better simplifier than the staff has then your simplifier might produce spurious failures while evaluating the correctness equation. Talk to us to rectify the situation and get some bonus marks.

	Shared	Secret
TestSimplifierEquivalence	30	10
TestSimplifier2	40	20

Note that you do not earn any marks for this lab until TestObjectContractF from lab3 passes. As you can see, the testing equations for this lab depend on having the object contract correctly implemented for the $\mathcal{F}$ ast classes. If your object contract implementation is buggy then we cannot trust the results of the testing equations.

Compiler Concepts: parser generators, Parsing Expression Grammars (PEG), push-down automata
Programming Concepts: domain specific languages (DSL): internal *vs.* external, debugging generated code, stacks

Lab 5
Parsing $\mathcal{W}$ with Parboiled

Files:
ece351.w.parboiled.WParboiledRecognizer
ece351.w.parboiled.WParboiledParser
Tests:
ece351.w.parboiled.TestWParboiledRecognizerAccept
ece351.w.parboiled.TestWParboiledRecognizerReject
ece351.w.parboiled.TestWParboiledParserAccept
ece351.w.parboiled.TestWParboiledParserReject
Tool Documentation:
http://parboiled.org

In this lab you will write a new parser for $\mathcal{W}$. In lab1 you wrote a parser for $\mathcal{W}$ by hand. In this lab you will write a parser for $\mathcal{W}$ using a parser generator tool named *Parboiled*. A parser generator is a tool that takes a description of a grammar and generates code that recognizes whether input strings conform to that grammar. Many developers in practice choose to use a parser generator rather than write parsers by hand.

There are many different parser generator tools, and there are a number of dimensions in which they differ, the two most important of which are the theory they are based on and whether they require the programmer to work in an *internal* or *external* DSL. The theory behind Parboiled is called *Parsing Expression Grammars* (PEG). Other common theories are LL (*e.g.*, Antlr) and LALR (*e.g.*, JavaCup).

DSL = Domain Specific Language. This terminology is in contrast to a *general purpose* programming language, such as Java/C/*etc.*

A DSL is often used in combination with a general purpose programming language, which is sometimes called the *host* language. For example, you might write an SQL query in a string inside a Java program. In this example Java is the host language and SQL is an external DSL for database queries. Whether the DSL is internal or external is determined by whether it shares the grammar of the host language, not by the physical location of snippets written in the DSL. In this example the SQL snippet is written inline in the Java program, but SQL has a different grammar than Java, and so is considered to be an external DSL.

An internal DSL uses the grammar of the host language. A common way to implement an internal DSL is as a library written in the host language. Not all libraries represent DSL's, but almost all internal DSL's are implemented as libraries.

Most parser generators require the programmer to work in an external DSL. That is, they require the programmer to learn a new language for specifying a grammar. Parboiled, by contrast, provides an internal DSL. I think that it is easier to learn an internal DSL, and this is why we have chosen to use Parboiled.

The Tiger Book §3.4 discusses two other parser generator tools, JavaCC and SableCC. You can look at that section to get a sense for what their external DSL's look like, and why it might be easier to learn Parboiled first. Parboiled did not exist when the Tiger book was written.

The purpose of this lab is for you to learn how to use Parboiled. This lab is specifically structured to facilitate this learning. You are already familiar with the input language $\mathcal{W}$ and the host language Java. $\mathcal{W}$ is an even simpler language than is used to teach Parboiled on its website. The only new thing in this lab is Parboiled.

Once you learn to use one parser generator tool it is not too hard to learn another one. Learning your first one is the most challenging.

The other reason why we are using Parboiled is that it is like a pushdown automata because its main storage mechanism is a stack, and so it reinforces that theoretical concept.

Learning progression: repetition with increasing complexity. In a traditional compilers project, without learning progression, you would have to learn everything simultaneously in one big lab: the source language ($\mathcal{W}$), the host language (Java), and the parser generator's external DSL. Moreover, the source language would be more like our toy subset of VHDL, which we won't see until after midterm week, rather than a simple regular language like $\mathcal{W}$. Even the pedagogical example on the Parboiled website is as complicated as $\mathcal{F}$ (a simple context-free language).

5.1 *Introducing Parboiled*

To write a recognizer or a parser with Parboiled we extend the BaseParser class, which either defines or inherits almost all of the methods that we will use from Parboiled. For our labs we will actually extend BaseParser351, which in turn extends BaseParser and adds some additional utility methods.

We can divide the methods available in our recognizer/parser into a number of groups:

Rule Constructors. These methods are used to describe both the grammar and the lexical specification of the language that we wish to recognize or parse, and we can subdivide this group this way:

EBNF	*Parboiled*	
*	ZeroOrMore()	
+	OneOrMore()	
?	Optional()	
|	FirstOf()	similar but importantly different
	Sequence()	no explicit character in EBNF

Regex	*Parboiled*	
[ab]	AnyOf("ab")	
[^ab]	NoneOf("ab")	
a	Ch('a') or just 'a'	
[a-z]	CharRange('a', 'z')	
	IgnoreCase()	no regex equivalent
	EOI	special char for end of input
	W0()	optional whitespace (zero or more)
	W1()	mandatory whitespace (at least one)

Recall that in previous labs there was a separate Lexer class that encoded the lexical specification for $\mathcal{W}$ (and $\mathcal{F}$). Some parser generator tools have separate DSL's for the lexical and syntactic specifications of the input language. In Parboiled, by contrast, we specify the tokenization as part of the grammar.

EBNF = Extended Backus Naur Form. This is the name of the notation used to specify the grammars for $\mathcal{W}$ (Figure 1.2) and $\mathcal{F}$ (Figure 3.1).

Regular expressions are often used for lexical specifications.

For this lab we will specify whitespace explicitly. In the next lab we will learn how to specify the whitespace implicitly, which makes the recognizer rules look a bit less cluttered.

Access to input. A recognizer doesn't need to store any of the input that it has already examined. A parser, however, often saves substrings of the input into an AST. The match() method returns the substring that matched the most recently triggered rule.

The match() method is in Parboiled's BaseActions class, which is the superclass of BaseParser. Draw a UML class diagram for WParboiledRecognizer and WParboiledParser.

Stack manipulation. A parser builds up an AST. Where do the fragments of this AST get stored while the parser is processing the input? When we wrote the parsers for $\mathcal{W}$ and $\mathcal{F}$ by hand in the previous labs we used a combination of fields, local variables, and method return values to store AST fragments. None of these storage mechanisms are available to us within Parboiled's DSL. Parboiled gives us one and only one place to store AST fragments: the Parboiled value stack. In this sense, working with Parboiled is very much like programming a pushdown automata. We can manipulate this stack with the standard operations, including: push(), pop(), peek(), swap(), and dup(). When the parser has processed all of the input we expect to find the completed AST on the top of the stack.

Hewlett-Packard calculators also famously have a value stack for intermediate results.

These stack operations are in Parboiled's BaseActions class, which is the superclass of BaseParser.

Grammar of the input language. We will add a method to our recognizer/parser for each of the non-terminals in the grammar of the input language. These methods will have return type Rule and will comprise just one statement: a return of the result of some Parboiled rule constructor.

We might also add some methods for parts of the lexical specification of the input language.

See rule constructors above.

5.2 *Write a recognizer for $\mathcal{W}$ using Parboiled*

Suppose we want to process files that contain a list of names, such as 'Larry Curly Moe '. A recognizer for such a language might include a snippet as in Figure 5.1.

```
public Rule List() { return ZeroOrMore(Sequence(Name(), W0())); }
public Rule Name() { return OneOrMore(Letter()); }
public Rule Letter() { return FirstOf(CharRange('A', 'Z'), CharRange('a', 'z')); }
```

Figure 5.1: Snippet of a recognizer written in Parboiled's DSL.

Write the EBNF grammar that corresponds to this Parboiled code.

5.3 *Add actions to recognizer to make a parser*

Let's add actions to our recognizer from Figure 5.1 to make a parser. Figure 5.2 lists code for the actions. The general idea is that we wrap every recognizer rule in a Sequence constructor, and then add new clauses to manipulate the stack: *i.e.*, a clause with one of the push, pop, peek, swap, dup, *etc.*, commands.

Figure 5.5 augments the listing of Figure 5.2 with some debugging clauses using the debugmsg() and checkType() methods. If you want to inspect memory while your recognizer or parser is executing then use one of the debugmsg() or checkType() methods and set a breakpoint inside that method. Setting a breakpoint inside a rule constructor will not do what you want.

debugmsg() and checkType() are defined in BaseParser351, which is a superclass of our recognizer/parser classes and a subclass of Parboiled's BaseParser.
Rule constructors are executed once in a grammar analysis phase in order to generate code that will actually process input strings.

```
public Rule List() {
    return Sequence(                                    // wrap everything in a Sequence
        push(ImmutableList.of()),                       // push empty list on to stack
        ZeroOrMore(Sequence(Name(), W0()))              // rule from recognizer
        );
    }
public Rule Name() {
    return Sequence(                                    // wrap everything in a Sequence
        OneOrMore(Letter()),                            // consume a Name, which can be retrieved later by match()
        push(match()),                                  // push the Name on top of the stack
        swap(),                                         // swap the Name and the List on the stack
        push( ((ImmutableList)pop()).append(pop()) )    // make a new list by appending the new Name to the old List
        );
}
public Rule Letter() { return FirstOf(CharRange('A', 'Z'), CharRange('a', 'z')); }    // rule from recognizer
```

Figure 5.2: Snippet of a parser written in Parboiled's DSL, corresponding to the recognizer snippet in Figure 5.1. See this snippet extended with some debugging commands in Figure 5.5.

When the parser reaches the end of the input then we expect to find a list object on the top of the stack, and we expect that list object to contain all of the names given in the input. For example, for the input string 'Ren Stimpy ' we would expect the result of the parse to be the list ['Ren', 'Stimpy']. Similarly, for the input string 'Larry Curly Moe ' we would expect the result of the parse to be the list ['Larry', 'Curly', 'Moe']. The result of a parse is the object on the top of the stack when the parser reaches the end of the input. Figure 5.3 illustrates the state of the stack as this example parser processes the input 'Ren Stimpy '.

http://www.youtube.com/results?search_query=ren+stimpy

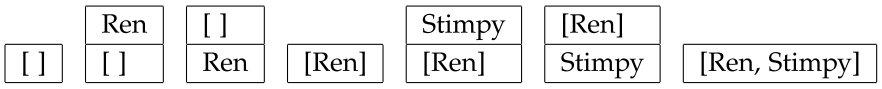


Figure 5.3: Stack of Parboiled parser from Figure 5.2 while processing input 'Ren Stimpy '. Time proceeds left to right. Think about which state of the stack corresponds to which line of the parser source code.

5.3.1 *An alternative Name() rule*

Figure 5.4 shows an alternative formulation of the Name() rule from Figure 5.2. Draw out the stack that this alternative formulation creates (*i.e.*, the analogue of Figure 5.3).

```
public Rule Name() {
    return Sequence(
        OneOrMore(Letter()),
        push( ((ImmutableList)pop()).append(match()) )
        );
}
```

Figure 5.4: An alternative definition of the Name() rule in Figure 5.2

```
public Rule List() {
    return Sequence(                                    // wrap everything in a Sequence
        push(ImmutableList.of()),                       // push empty list on to stack
        ZeroOrMore(Sequence(Name(), W1()))              // rule from recognizer
        checkType(peek(), List.class)                   // expect a list on the top of the stack
        );
    }
public Rule Name() {
    return Sequence(                                    // wrap everything in a Sequence
        OneOrMore(Letter()),                            // rule from recognizer
        push(match())                                   // push a single name, so stack is: [List, String]
        debugmsg(peek()),                               // print the matched name to System.err
        checkType(peek(), String.class)                 // match always returns a String
        swap(),                                         // swap top two elements on stack: [String, List]
        checkType(peek(), List.class)                   // confirm that the list is on top
        push( ((ImmutableList)pop()).append(pop()) )    // construct and push a new list that contains
                                                        // all of the names on the old list plus this new name
        );
}
public Rule Letter() { return FirstOf(CharRange('A', 'Z'), CharRange('a', 'z')); }    // rule from recognizer
```

Figure 5.5: Snippet of a parser written in Parboiled's DSL (Figure 5.2). Extended with debugging commands. Corresponding to the recognizer snippet in Figure 5.1.

5.3.2 *The stack for our $\mathcal{W}$ parser will have two objects*

Draw out a picture of what you expect the stack to look like at each point in time of the parser execution *before you start programming it*. See Figure 5.3 as an example.

What is the maximum size that the stack will grow to?

The stack for our $\mathcal{W}$ programs will have two objects once it reaches steady state: a WProgram object and a Waveform object. Periodically the Waveform object will be appended with the WProgram object and a new Waveform object will be instantiated. The WProgram object will usually be closer to the bottom of the stack and the Waveform object will usually be closer to the top.

5.4 *Parsing Expression Grammars (PEG's)*†

Parboiled works with *Parsing Expression Grammars* (PEG's). PEG's are slightly different than *Context Free Grammars* (CFG's). The important difference is that PEG's cannot represent ambiguity, whereas CFG's can. We never want ambiguity in programming languages, so this 'feature' of CFG's is not much of a feature in our context.

http://en.wikipedia.org/wiki/Parsing_expression_grammar

http://en.wikipedia.org/wiki/Context-free_grammar

CFG's were originally developed by linguists to model natural languages, which often contain grammatical ambiguities. Only after linguists had successfully applied the idea of CFG's in their natural language context did computer scientists try using CFG's for programming languages.

Where does ambiguity arise in a CFG? From the choice (alternation) operator: '|' (the bar). An ambiguity is when two (or more) different alternatives might apply. PEG's remove this possibility for ambiguity by treating alternatives in priority order: the first alternative to match is taken as correct; if a subsequent alternative would also have matched it is just ignored. So Parboiled does not have a bar operator, it has the FirstOf method instead, which says 'use the alternative in this list that matches first.'

CFG's and PEG's recognize almost the same set of grammars, but there are a few that each can do that the other cannot. For example, a CFG can recognize the following, while a PEG cannot:
$S \rightarrow\ 'x'\ S\ 'x'\ |\ 'x'$
Similarly, there are some grammars that PEG's can recognize that CFG's cannot, such as counting three things:
$\{a^n b^n c^n | n \geq 0\}$

5.5 A few debugging tips

You might get a weird error message that Parboiled has difficulty creating the parser class. If so, see if your rule constructors are throwing exceptions. For example, the skeleton code ships with stubs like this:

You need to get rid of all of these stubs and replace them with code that does something sensible — or just return null. If you leave the exception throwing in, even in a rule constructor you don't call, Parboiled will complain.

```
public Rule Program() {
    // TODO: 1 lines snipped
    throw new ece351.util.Todo351Exception();
}
```

Also, you should explicitly include EOI in your grammar.

5.6 Evaluation

The last pushed commit before the deadline is evaluated both on the shared test inputs and on a set of secret test inputs according to the weights in the table below.

You do not earn any marks for the rejection tests until some acceptance tests pass. You do not earn any marks for the main parser tests until the basic parser test passes.

	Current	New
TestWParboiledRecognizerAccept	20	5
TestWParboiledRecognizerReject	5	5
TestWParboiledParserBasic	5	0
TestWParboiledParserAccept	30	15
TestWParboiledParserReject	10	5

The recognizer tests just run the recognizer to see whether it rejects or accepts. The parser rejection tests work similarly. The parser acceptance tests check two testing equations. Let w name the input $\mathcal{W}$ file. Let x = ParboiledParse(w). The two testing equations are:

```
x.isomorphic(ParboiledParse(PrettyPrint(x)))
x.isomorphic(RecursiveDescentParse(w))
```

5.7 Reading

Parboiled & peg's:
http://parboiled.org
https://github.com/sirthias/parboiled/wiki/Grammar-and-Parser-Debugging
https://github.com/sirthias/parboiled/wiki/Handling-Whitespace
http://www.decodified.com/parboiled/api/java/org/parboiled/BaseParser.html
http://en.wikipedia.org/wiki/Parsing_expression_grammar

JavaCC & SableCC: (two other parser generators)
Tiger Book §3.4
Note that both JavaCC and SableCC require the grammar to be defined in an external dsl (*i.e.*, not in Java).

You will not be tested on specifics of JavaCC and SableCC, but you should know of their existence and that they use an external dsl for specifying the grammar.

Lab 6
Parsing $\mathcal{F}$ with Parboiled

Compiler Concepts: parser generators, Parsing Expression Grammars (PEG), push-down automata

Programming Concepts: domain specific languages (DSL): internal *vs.* external, debugging generated code, stacks

Files:
ece351.f.parboiled.FParboiledRecognizer.java
ece351.f.parboiled.FParboiledParser.java

Tests:
ece351.f.parboiled.TestFParboiledRecognizerAccept
ece351.f.parboiled.TestFParboiledRecognizerReject
ece351.f.parboiled.TestFParboiledParserBasic
ece351.f.parboiled.TestFParboiledParser
ece351.f.parser.TestFParserComparison

This lab is similar to some previous labs: we will write a recognizer and parser for $\mathcal{F}$, this time using Parboiled.

6.1 *Write a recognizer for $\mathcal{F}$ using Parboiled*

Program	$\rightarrow$	Formula$^+$
Fomula	$\rightarrow$	Var '<=' Expr ';'
Expr	$\rightarrow$	Term ('or Term)*
Term	$\rightarrow$	Factor ('and' Factor)*
Factor	$\rightarrow$	'not' Factor | '(' Expr ')' | Var | Constant
Constant	$\rightarrow$	'0' | '1'
Var	$\rightarrow$	id

Figure 6.1: LL(1) Grammar for $\mathcal{F}$ (reproduced from Figure 3.1)

INSPECT TOKEN WITHOUT CONSUMING IT. Parboiled has two rule constructors that allow you to inspect the next token without consuming it: Test() and TestNot. For example, in a rule for Var one might include TestNot(Keyword()) to ensure that a keyword (*e.g.*, 'and', 'or') is not considered as a variable name.

IMPLICIT WHITESPACE HANDLING. As we saw in a previous lab, Parboiled incorporates both syntactic and lexical specifications. One of the practical consequences of this is that we cannot delegate whitespace handling to a separate lexer (as we did when writing recursive descent parsers by hand). The explicit mention of whitespace after every literal can make the recognizer/parser code look cluttered. There is a standard trick in Parboiled that if we add a single trailing space to each literal then the match for that literal will include all trailing whitespace. For example, in the rule constructor for Constant we would write: FirstOf("0 ", "1 ") (notice the trailing space after the zero and the the one). If you look at the method ConstantExpr.make(s) you will see that it looks only at the first character of its argument s, and so thereby ignores any trailing whitespace. When you call match() the result will contain the trailing whitespace.

If you are interested in how this trick works there is a page devoted to it on the Parboiled website. This is just a trick of the tool and not part of the intellectual content of this course, so you are not expected to understand how it works.

6.2 Add actions to recognizer to make a parser

Following our standard design method, once your recognizer is working, copy it to start making the parser. As before, you will introduce a new Sequence at the top level of each rule constructor, and the parser actions will be additional arguments to this Sequence.

We will reuse the $\mathcal{F}$ AST classes from the previous lab.

We highly recommend that you sketch out what you expect the Parboiled stack to look like as it parses three different inputs. Once you have a clear understanding of what the stack is supposed to look like then writing the code is fairly easy.

See Figure 5.3 for an example of a sketch of the state of the stack.

How large can the stack grow while parsing $\mathcal{F}$?

Your Parboiled $\mathcal{F}$ parser should construct BinaryExprs, just as your recursive descent $\mathcal{F}$ parser did. We can apply the same set of transformations, developed in a previous lab, to produce NaryExprs.

6.3 Introducing the Composite Design Pattern†

The Composite design pattern is commonly used when we need to construct trees of objects, and we want to work with the leaf nodes and the interior nodes ('composite' nodes) via a common interface ('component'). Figure 6.2 illustrates this idea.

The main place we've seen the composite design pattern in the labs is the $\mathcal{F}$ AST classes. The classes representing the leaves of an AST are ConstantExpr and VarExpr. The classes representing the interior ('composite') nodes of the AST are AndExpr, OrExpr, NotExpr, *etc.*. The shared interface is Expr ('component').

What about classes like BinaryExpr and UnaryExpr?

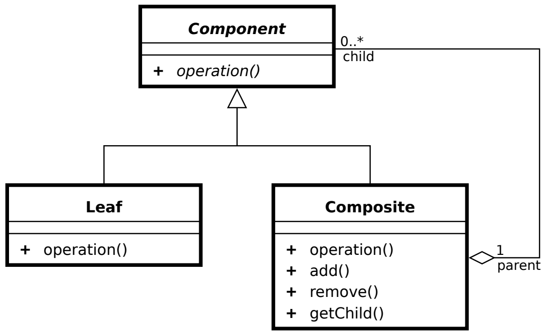


Figure 6.2: UML class diagram for Composite Design Pattern.
From http://en.wikipedia.org/wiki/File:Composite_UML_class_diagram_(fixed).svg, where it is released in the public domain (*i.e.*, free for reuse).

Why didn't we use the Composite design pattern for the $\mathcal{W}$ AST classes? Because $\mathcal{W}$ is a regular language, and so does not have nested expressions, it will always result in AST's of a known fixed height. $\mathcal{F}$, on the other hand, is a context-free language with nested expressions, so the AST depth will vary from input to input. The Composite design pattern lets us work with these AST's of varying depth in a uniform way.

6.4 Introducing the Singleton Design Pattern†

http://en.wikipedia.org/wiki/Singleton_pattern

The singleton design pattern is used when the number of objects to be instantiated for a given class is independent of the program's input. If the singleton objects are immutable then the primary motivation is to conserve memory by not creating duplicate objects. If the singleton objects are mutable then the primary motivation is to create a set of global variables (which is a controversial decision[1]).

The main use of the singleton pattern in our labs is the ConstantExpr class, which is immutable, so the motivation is to conserve memory by instantiating just one object for True and one for False, regardless of how many times '1' and '0' appear in the $\mathcal{F}$ programs manipulated by the compiler. Figure 6.3 shows the creation of the two instances of class ConstantExpr.

[1] W. Wulf and M. Shaw. Global variables considered harmful. *ACM SIGPLAN Notices*, 8(2):80–86, Feb. 1973
http://c2.com/cgi/wiki?GlobalVariablesConsideredHarm

```
 */
public final class ConstantExpr extends Expr {
        public final Boolean b;

        /** The one true instance. To be shared/aliased wherever necessary. */
        public final static ConstantExpr TrueExpr = new ConstantExpr(true);
        /** The one false instance. To be shared/aliased wherever necessary. */
```

Figure 6.3: The only two instances of class ConstantExpr

While Figure 6.3 shows the creation of two objects that have global names so they can be used by anyone, it does not prevent anyone from creating more objects of class ConstantExpr. To do that we make the constructor private, and then provide an alternative method to return a reference to one of the existing objects, as listed in Figure 6.4.

```
        /** Private constructor prevents clients from instantiating. */
        private ConstantExpr(final Boolean b) { this.b = b; }

        /** To be used by clients instead of the constructor.
          * Returns a reference to one of the shared objects. */
        public static ConstantExpr make(final Boolean b) {
                if (b) { return TrueExpr; } else { return FalseExpr; }
```

Figure 6.4: Preventing clients from instantiating class ConstantExpr

6.5 *Evaluation*

We evaluate the code that you have committed and pushed before the deadline. The test harnesses are run with the $\mathcal{F}$ programs we have released to you and a secret set of staff $\mathcal{F}$ programs. The weights for the test harnesses are as follows:

	Current	New
TestFParboiledRecognizerAccept	10	5
TestFParboiledRecognizerReject	5	0
TestFParboiledParserBasic	5	0
TestFParboiledParser	40	10
TestFParboiledParserComparison	20	5

You do not get any marks for any other parser tests until TestFParboiledParser-Basic and TestObjectContractF pass everything.

TestFParboiledRecognizer just runs the recognizer to see if it crashes. TestFParboiledParser checks the following equation:

∀ AST | AST.equals(parse(prettyprint(AST)))

TestFParboiledParserComparison checks that your recursive descent and Parboiled parsers give isomorphic AST's when given the same input file.

Lab 7
Technology Mapping: $\mathcal{F} \rightarrow$ Graphviz

Compiler Concepts: common subexpression elimination

Programming Concepts: hash structures, iteration order, object identity, non-determinism, Visitor design pattern, tree traversals: preorder, postorder, inorder

Files:
ece351.f.TechnologyMapper

Libraries:
ece351.f.ExtractAllExprs
ece351.common.visitor.PostOrderExprVisitor

Tests:
ece351.f.TestTechnologyMapper

In this lab we will produce gate diagrams of our $\mathcal{F}$ programs. We will do this by translating $\mathcal{F}$ programs into the input language of AT&T GraphViz. Graphviz is an automatic graph layout tool: it reads (one dimensional) textual descriptions of graphs and produces (two dimensional) pictures of those graphs. Graphviz, which is primarily a visualization tool, attempts to place nodes and edges to minimize edge crossings. Tools that are intended to do digital circuit layout optimize the placement of components based on other criteria, but the fundamental idea is the same: transform a (one dimensional) description of the logic into a (two dimensional) plan that can be visualized or fabricated.

http://graphviz.org

Graphviz is a widely used graph layout tool. Circuit layout is often done by specialized tools, but the basic idea is the same as Graphviz: algorithmic placement of interconnected components. What differs is the criteria used for placement and the ways in which things are connected. Graphical layout tools such as Graphviz try to minimize edge crossings, for example, which is important for people looking at pictures but may be less relevant for circuit boards. Graphviz also draws nice Bezier curve lines that are pleasing to look at, whereas circuit boards are typically laid out with orthogonal lines. Technology mapping and circuit layout are studied in ECE647.

Figure 7.1 lists a sample Graphviz input file and its rendered output for the simple $\mathcal{F}$ program `X <= A or B;`. The input pins `A` and `B` are on the left side of the diagram and the output pin `X` is on the right hand side of the diagram.

```
digraph g {
    // header
    rankdir=LR;
    margin=0.01;
    node [shape="plaintext"];
    edge [arrowhead="diamond"];
    // circuit
    or12 [label="or12", image="../../gates/or_noleads.png"];
    var0[label="x"];
    var1[label="a"];
    var2[label="b"];
    var1 −> or12 ;
    var2 −> or12 ;
    or12 −> var0 ;
}
```

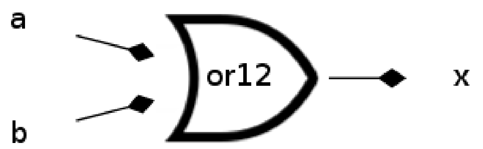


Figure 7.1: Example Graphviz input file and rendered output for $\mathcal{F}$ program `X <= A or B;`

The input file in Figure 7.1 contains a header section that will be common to all of the Graphviz files that we generate. This header says that the graph should be rendered left to right (instead of the default top-down), that the whitespace margin around the diagram should be 0.01 inches, and that the default look for nodes and edges should be plain.

The main thing to notice in the lines of the input file in Figure 7.1 that describe the formula is that visual nodes are created for both the pins (A, B, X) and the gates (there is a single OR gate in this example).

7.1 *Interpreters* vs. *Compilers*†

An *programming language interpreter* is a special kind of compiler that executes a program directly, instead of translating the program to another language that is then executed. Executing a program through a language interpreter is usually slower than executing the compiled form of that program. It can take less engineering effort to implement a language interpreter than a compiler, so language interpreters are most commonly used when engineering time is more valuable than machine time. Compilers are used when the converse is true: *i.e.,* it is worth the engineering effort to make the code run faster.

7.2 *Introducing the Interpreter Design Pattern*†

The *Interpreter design pattern*[1] names a particular way to structure code, that we have been using in previous labs but have not yet named. Programming language interpreters are often implemented using the interpreter design pattern, which is how the design pattern got its name. When learning about both concepts this name collision can be a bit confusing.

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass., 1995

Figure 7.2 shows a UML class diagram for a subset of the $\mathcal{F}$ expression class hierarchy that we've been working with recently.

In Lab 4 we wrote a simplifier/optimizer for $\mathcal{F}$ programs. How did we structure that code? We added a method called simplify() to almost every class in this hierarchy, similar to what is depicted in Figure 7.3. This is a non-modular change: the simplifier functionality is scattered across the Expr class hierarchy.

Is it possible to structure this code in another way? Could we add the simplifier functionality to our compiler in a modular way?

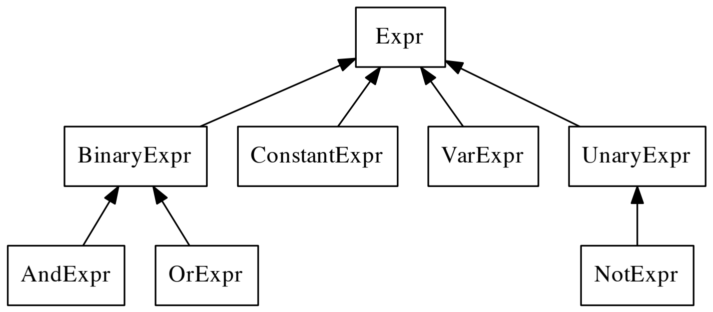


Figure 7.2: A UML class diagram for a subset of $\mathcal{F}$ expression class hierarchy

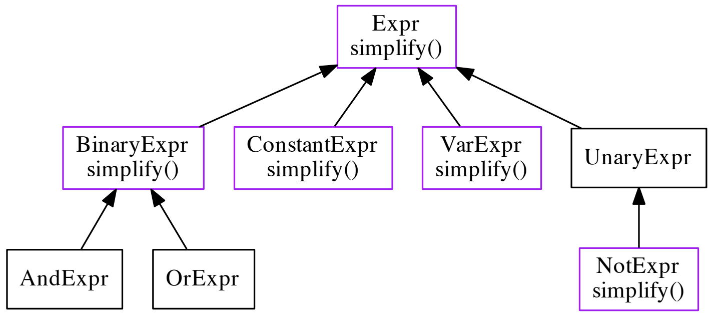


Figure 7.3: A UML class diagram for a subset of $\mathcal{F}$ expression class hierarchy, showing the simplifier functionality scattered across the hierarchy by use of the Interpreter design pattern. Classes modified to add the simplifier functionality are highlighted in colour.

	Operation		
AST Class	simplify	pretty-printing	conversion to gates
AndExpr	AndExpr (BinaryExpr)	AndExpr (BinaryExpr)	AndExpr (BinaryExpr)
OrExpr	OrExpr (BinaryExpr)	OrExpr (BinaryExpr)	OrExpr (BinaryExpr)
VarExpr	VarExpr	VarExpr	VarExpr
ConstantExpr	ConstantExpr	ConstantExpr	ConstantExpr
NotExpr	NotExpr	NotExpr	NotExpr
NaryAndExpr	NaryAndExpr (NaryExpr)	NaryAndExpr (NaryExpr)	NaryAndExpr (NaryExpr)
NaryOrExpr	NaryOrExpr (NaryExpr)	NaryOrExpr (NaryExpr)	NaryOrExpr (NaryExpr)

Figure 7.4: Matrix view of Interpreter Design Pattern. Rows for AST leaf classes. Columns for operations to be performed on the AST. Cell values indicate where the code for that class/op pair is located. In some cases we place the operation in an abstract superclass, *e.g.*, BinaryExpr, rather than directly in the leaf class. This location is indicated with the concrete leaf class name followed by the abstract superclass name in parenthesis. Notice that the pattern here is that the same value is repeated across each row.

7.3 Introducing the Visitor Design Pattern†

Tiger Book §4.3

The *Visitor* design pattern[2] is an alternative way to structure code from the interpreter design pattern. The Visitor pattern is also widely used in compilers, and is how we will structure our code in this lab and future labs. If we had used the Visitor design pattern for Lab 4 then we could have added the simplifier functionality in a more modular manner, as depicted in Figure 7.5.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass., 1995

Also widely documented on the web.

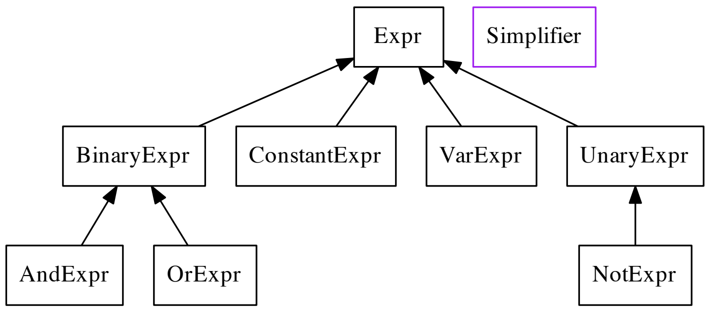


Figure 7.5: A UML class diagram for a subset of $\mathcal{F}$ expression class hierarchy, showing the simplifier functionality made modular by use of the Visitor design pattern.

Well, it's not quite as simple as Figure 7.5 might lead you to believe: there is some infrastructure that we need to add to make this work properly. Figure 7.6 illustrates that we need to add an accept(Visitor) method to each concrete class.[3] Similarly, we add a visit() for each concrete class to the Simplifier class. These accept(Visitor) methods can be reused by any Visitor, whether it is the simplifier or the technology mapper that we will develop in this lab.

[3] Notice that in our $\mathcal{F}$ Expr class hierarchy the only classes that can be directly instantiated (*i.e.*, are concrete) are the leaves of the hierarchy. This is sometimes known as the *abstract superclass rule* (*i.e.*, all super-classes should be abstract), and is a good design guideline for you to follow.

With the use of the Visitor design pattern we can add new operations (*e.g.*, simplification, technology mapping, *etc.*) to our complex data structure (AST) in a modular manner, without having to directly modify the AST classes.

The Visitor design pattern involves two main methods: *visit* and *accept*. The *visit* methods are written on the Visitor class and the *accept* methods are written on the AST classes. The abstract superclass for the Visitors that we will write is listed in Figure 7.8.
Notice that our visit methods return an object of type Expr. This feature allows our Visitors to rewrite the AST as they visit it, which is necessary for transformations such as the simplifier we wrote previously. The Visitor we write in this lab does not rewrite the AST, and so all of its visit methods will simply return the AST node that they are visiting (*i.e.*, return e;).

To the FExpr class we add the signature for the accept method,

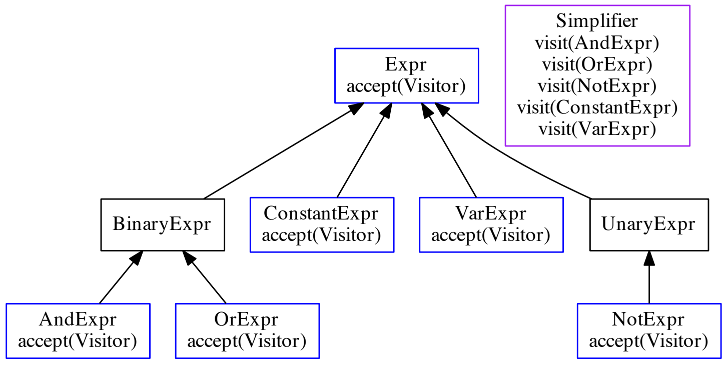


Figure 7.6: A UML class diagram for a subset of $\mathcal{F}$ expression class hierarchy, showing the simplifier functionality made modular by use of the Visitor design pattern.

	Operation		
AST Class	simplify	pretty-printing	conversion to gates
AndExpr	Simplifier	PPrinter	TechMapper
OrExpr	Simplifier	PPrinter	TechMapper
VarExpr	Simplifier	PPrinter	TechMapper
ConstantExpr	Simplifier	PPrinter	TechMapper
NotExpr	Simplifier	PPrinter	TechMapper
NaryAndExpr	Simplifier	PPrinter	TechMapper
NaryOrExpr	Simplifier	PPrinter	TechMapper

Figure 7.7: Matrix view of the Visitor Design Pattern. Rows for AST leaf classes. Columns for operations to be performed on the AST. Cell values indicate where the code for that class/op pair is located. Notice that the pattern here is that the values are repeated down each column, rather than across each row as we saw above in Figure 7.4 for the Interpreter Design Pattern.

In the actual code for our labs we followed the Interpreter Design Pattern for the simplify and pretty-printing operations, so the class names Simplifier and PPrinter are fictitious. We are developing the TechnologyMapper visitor in this lab.

```
public abstract class ExprVisitor {
    public abstract Expr visitConstant(ConstantExpr e);
    public abstract Expr visitVar(VarExpr e);
    public abstract Expr visitNot(NotExpr e);
    public abstract Expr visitAnd(AndExpr e);
    public abstract Expr visitOr(OrExpr e);
    public abstract Expr visitNaryAnd(NaryAndExpr e);
    public abstract Expr visitNaryOr(NaryOrExpr e);
```

Figure 7.8: Abstract super class for Visitors for $\mathcal{F}$ expression ASTs. There is a visit method for each concrete $\mathcal{F}$ AST class.

which is then implemented in each concrete AST class as a call to a visit method, as shown in Figure 7.9.

```
public abstract Expr accept(final ExprVisitor exprVisitor);

public Expr accept(final ExprVisitor v) { return v.visitAnd(this); }
```

Figure 7.9: Signature and implementation of accept method. The signature belongs in the Expr class and the implementation is put in each concrete subclass of Expr.

Together the visit and accept methods implement what is known as *double dispatch: i.e.*, select which method to execute based on the polymorphic type of two objects. Languages like Java, C++, C#, and Objective C are all *single dispatch* languages: the target of a polymorphic call is determined just by the type of the receiver object. CLOS and Dylan are examples of *multiple-dispatch* languages, where the target of a polymorphic call is determined by the runtime types of all of arguments.

One of the main points of variation in how the Visitor pattern is implemented is where the traversal code goes: in the visit methods? an iterator? somewhere else? All of these options are used in practice. We have decided to put the traversal code in a set of *traverse* methods in the Visitor classes (Figures 7.10 and 7.11).

The point of a Visitor is to *traverse*, or 'walk over', the nodes in an AST.

We can write Visitors that perform a number of useful tasks for this lab. For example, Figure 7.12 lists a Visitor that builds a set of all the nodes in an $\mathcal{F}$ expression AST.

7.3.1 *Is Visitor always better than Interpreter?*†

From the discussion above you might think that it is always better to use the Visitor pattern than the Interpreter pattern. This is not correct. Which pattern to choose depends on what changes you anticipate happening to the software in the future.

Almost all software evolves or dies. Donald Knuth's TeX is a notable counter-example: it does not grow new features, it does not have significant bugs, it was conceived complete. Consequently, the version number for TeX does not increment, it converges: as the (minor) bugs are removed the version number approximates π by another decimal place. This reflects convergence to an original vision, with the acknowledgement that perfection will never truly be achieved within the finite time available to us here on earth.

Consider this: we have a complex structure (*e.g.*, an AST), and a set of complex operations that we would like to apply to that structure (*e.g.*, simplification, technology mapping, *etc.*). In ECE250 you studied some relatively simple structures, such as lists, and some relatively simple operations, such as searching or sorting. Those structures were simple enough that they could be defined in one or two classes, and those operations were simple enough that they could be defined in one or two methods. Now we have a complex structure that is defined across a dozen or more classes, and operations that might similarly be defined across a dozen or more methods.

The choice of how we organize this code (Visitor or Interpreter) has consequences, and those consequences are in how difficult the code will be to change in the future.

If we expect the grammar of our input language (*e.g.*, $\mathcal{W}$ or $\mathcal{F}$) to be stable, and if we expect to add more transformations (operations) to the AST in future labs, then it is better to use the Visitor pattern, because the Visitor pattern allows us to add new transformations in a modular fashion.

If, on the other hand, we expect the grammar of our input language to change and the set of transformations we might want to perform is small and known *a priori*, then it is better to use Interpreter. Interpreter lets us change the AST class hierarchy in a modular fashion, which is what we would need to do if the grammar of the input language were to change significantly.

What if we anticipate both kinds of change will happen in the future? Then we have a problem, because there is no widely accepted programming language that lets us structure our code to facilitate both kinds of change. This is known as the *Expression Problem* in programming language design. The Expression Problem was named by Wadler in the late nineties[4], although the idea goes back at least as far as Reynolds work in the mid seventies[5]. More recent research papers have proposed some solutions in Java[6] or Scala[7], but none of these proposals is widely accepted and all have drawbacks.

These references below are for your broader educational enrichment. You will not be tested on them specifically. You are expected to understand the trade-offs between the Visitor and Interpreter design patterns.

[4] P. Wadler. The expression problem, Nov. 1998. Email to Java Generics list

[5] J. C. Reynolds. User-defined types and procedural data as complementary approaches to data abstraction. In S. A. Schuman, editor, *New Directions in Algorithmic Languages: IFIP Working Group 2.1 on Algol*. INRIA, 1975

[6] M. Torgersen. The Expression Problem Revisited: Four new solutions using generics. In M. Odersky, editor, *Proc.18th ECOOP*, volume 3344 of *LNCS*, Oslo, Norway, June 2004. Springer-Verlag

[7] M. Zenger and M. Odersky. Independently extensible solutions to the expression problem. In *Proc.12th Workshop on Foundations of Object-Oriented Languages*, 2005

Figure 7.10: The traverse methods of ExprVisitor

```
/**
 * Dispatch to appropriate traverse method.
 */
public final Expr traverseExpr(final Expr e) {
        if (e instanceof NaryExpr) {
                return traverseNaryExpr( (NaryExpr) e );
        } else if (e instanceof BinaryExpr) {
                return traverseBinaryExpr( (BinaryExpr) e );
        } else if (e instanceof UnaryExpr) {
                return traverseUnaryExpr( (UnaryExpr) e );
        } else {
                return e.accept(this);
        }
}

public abstract Expr traverseNaryExpr(final NaryExpr e);
public abstract Expr traverseBinaryExpr(final BinaryExpr e);
public abstract Expr traverseUnaryExpr(final UnaryExpr e);

/**
 * Visit/rewrite all of the exprs in this FProgram.
 * @param p input FProgram
 * @return a new FProgram with changes applied
 */
public FProgram traverseFProgram(final FProgram p) {
        FProgram result = new FProgram();
        for (final AssignmentStatement astmt : p.formulas) {
                result = result.append(traverseAssignmentStatement(astmt));
        }
        return result;
}
```

Figure 7.11: Implementation of PostOrderExprVisitor. Any Visitors that extend this class will visit the nodes of an $\mathcal{F}$ expression AST in post-order (*i.e.,* parents after children).

```
/**
 * This visitor rewrites the AST from the bottom up.
 * Optimized to only create new parent nodes if children have changed.
 */
public abstract class PostOrderExprVisitor extends ExprVisitor {

        @Override
        public final Expr traverseUnaryExpr(UnaryExpr u) {
                // child first
                final Expr child = traverseExpr(u.expr);
                // only rewrite if something has changed
                if (child != u.expr) {
                        u = u.newUnaryExpr(child);
                }
                // now parent
                return u.accept(this);
        }

        @Override
        public final Expr traverseBinaryExpr(BinaryExpr b) {
                // children first
                final Expr left = traverseExpr(b.left);
                final Expr right = traverseExpr(b.right);
                // only rewrite if something has changed
                if (left != b.left || right != b.right) {
                        b = b.newBinaryExpr(left, right);
                }
                // now parent
                return b.accept(this);
        }

        @Override
        public final Expr traverseNaryExpr(NaryExpr e) {
                // children first
                ImmutableList<Expr> children = ImmutableList.of();
                boolean change = false;
                for (final Expr c1 : e.children) {
                        final Expr c2 = traverseExpr(c1);
                        children = children.append(c2);
                        if (c2 != c1) { change = true; }
                }
                // only rewrite if something changed
                if (change) {
                        e = e.newNaryExpr(children);
                }
                // now parent
                return e.accept(this);
        }
}
```

Figure 7.12: Implementation of ExtractAllExprs

```
/**
 * Returns a set of all Expr objects in a given FProgram or AssignmentStatement.
 * The result is returned in an IdentityHashSet, which defines object identity
 * by memory address. A regular HashSet defines object identity by the equals()
 * method. Consider two VarExpr objects, X1 and X2, both naming variable X. If
 * we tried to add both of these to a regular HashSet the second add would fail
 * because the regular HashSet would say that it already held a VarExpr for X.
 * The IdentityHashSet, on the other hand, will hold both X1 and X2.
 */
public final class ExtractAllExprs extends PostOrderExprVisitor {

	private final IdentityHashSet<Expr> exprs = new IdentityHashSet<Expr>();

	private ExtractAllExprs(final Expr e) { traverseExpr(e); }

	public static IdentityHashSet<Expr> allExprs(final AssignmentStatement f) {
		final ExtractAllExprs cae = new ExtractAllExprs(f.expr);
		return cae.exprs;
	}

	public static IdentityHashSet<Expr> allExprs(final FProgram p) {
		final IdentityHashSet<Expr> allExprs = new IdentityHashSet<Expr>();
		for (final AssignmentStatement f : p.formulas) {
			allExprs.add(f.outputVar);
			allExprs.addAll(ExtractAllExprs.allExprs(f));
		}
		return allExprs;
	}

	@Override public Expr visitConstant(ConstantExpr e) { exprs.add(e); return e; }
	@Override public Expr visitVar(VarExpr e) { exprs.add(e); return e; }
	@Override public Expr visitNot(NotExpr e) { exprs.add(e); return e; }
	@Override public Expr visitAnd(AndExpr e) { exprs.add(e); return e; }
	@Override public Expr visitOr(OrExpr e) { exprs.add(e); return e; }
	@Override public Expr visitXOr(XOrExpr e) { exprs.add(e); return e; }
	@Override public Expr visitNAnd(NAndExpr e) { exprs.add(e); return e; }
	@Override public Expr visitNOr(NOrExpr e) { exprs.add(e); return e; }
	@Override public Expr visitXNOr(XNOrExpr e) { exprs.add(e); return e; }
	@Override public Expr visitEqual(EqualExpr e) { exprs.add(e); return e; }
	@Override public Expr visitNaryAnd(NaryAndExpr e) { exprs.add(e); return e; }
	@Override public Expr visitNaryOr(NaryOrExpr e) { exprs.add(e); return e; }
}
```

7.4 *Hash Structures, Iteration Order, and Object Identity*†

When you iterate over the elements in a List you get them in the order that they were added to the List. When you iterate over the elements in a TreeSet you get them sorted lexicographically. When you iterate over the elements in a HashSet or HashMap, what order do you get them in? Unspecified, unknown, and non-deterministic: the order could change the next time you iterate, and will likely change the next time the program executes.

TreeSet, List, HashSet, and HashMap are all part of the standard JDK Collections classes.

```
List list = new ArrayList();
list.add(3);
list.add(1);
list.add(2);
System.out.println(list);
```

```
SortedSet tset = new TreeSet();
tset.add(3);
tset.add(1);
tset.add(2);
System.out.println(tset);
```

```
Set hset = new HashSet();
hset.add(3);
hset.add(1);
hset.add(2);
System.out.println(hset);
```

Figure 7.13: Iteration order for different data structures

Why might the iteration order change with hash structures? Because the slot into which an element gets stored in a hash structure is a function of that element's hash value and *the size of the hash table*. As more elements are added to a hash structure then it will resize itself and rehash all of its existing elements and they'll go into new slots in the new (larger) table. If the same data value always produces the same hash value and the table never grows then it is possible to get deterministic iteration order from a hash structure — although that order will still be nonsense, it will be deterministically repeatable nonsense. But these assumptions often do not hold. For example, if a class does not implement the equals() and hashCode() methods then its memory address is used as its hashCode(). The next execution of the program is highly likely to put that same data value at a different memory address.

Iterating over the elements in a hash structure is one of the most common ways of unintentionally introducing non-determinism into a Java program. Non-determinism makes testing and debugging difficult because each execution of the program behaves differently. So unless there is some benefit to the non-determinism it should be avoided.

What benefit could there be to non-determinism? Not much directly. But non-determinism is often a consequence of parallel and distributed systems. In these circumstances we sometimes choose to tolerate some non-determinism for the sake of performance — but we still try to control or eliminate some of the non-determinism using mechanisms like locks or database engines.

The JDK Collections classes provide two hash structures with deterministic iteration ordering: LinkedHashSet and LinkedHashMap. These structures also maintain an internal linked list that records the order in which elements were added to the hash structure. You should usually choose LinkedHashMap, LinkedHashSet, TreeMap, or TreeSet over HashMap and HashSet. The linked structures give elements in their insertion order (as a list would), whereas the tree structures give you elements in alphabetical order (assuming that there is some alphabetical ordering for the elements).

You could safely use HashMap and HashSet without introducing non-determinism into your program *if you never iterate over their elements*. It's hard to keep that promise though. Once you've got a data structure you might want to print out its values, or pass it in to some third party code, *etc.* So it's better to just use a structure with a deterministic iteration ordering.

The skeleton code for this lab makes use of two other hash structures: IdentityHashMap and IdentityHashSet. What distinguishes these structures from the common structures?

IdentityHashMap is from the JDK. IdentityHashSet is from the open-source project named Kodkod. There are a number of open source projects that implement an IdentityHashSet due to its omission in the JDK. See Java bug report #4479578.

The definition of a set is that it does not contain duplicate elements. How is 'duplicate' determined? We work with four different definitions of 'duplicate' in these labs:

x == y x and y have the same physical memory address.

x.equals(y) any computation that uses x or y will have no observable differences.

x.isomorphic(y) x and y might have some minor structural differences, but are essentially the same.

x.equivalent(y) x and y are semantically equivalent and might not have any structural similarities.

```
Set hset = new HashSet();
hset.add(new VarExpr("Y"));
hset.add(new VarExpr("X"));
hset.add(new VarExpr("X"));
System.out.println(hset);
```

```
Set lhset = new LinkedHashSet();
lhset.add(new VarExpr("Y"));
lhset.add(new VarExpr("X"));
lhset.add(new VarExpr("X"));
System.out.println(lhset);
```

```
Set iset = new IdentityHashSet();
iset.add(new VarExpr("Y"));
iset.add(new VarExpr("X"));
iset.add(new VarExpr("X"));
System.out.println(iset);
```

Figure 7.14: Object identity for different data structures

The common structures (HashSet, TreeSet, *etc.*) use the equals() method to determine duplicates, whereas the IdentityHashSet and IdentityHashMap use the memory address (==) to determine duplicates. In this lab we want to ensure that our substitution table contains an entry for every single FExpr object (physical memory address), so we use IdentityHashSet and IdentityHashMap. Notice that ExtractAllExprs also returns an IdentityHashSet.

The skeleton code is careful about its use of IdentityHashSet/Map, LinkedHashSet/Map, and TreeSet/Map. You should think about the concept of duplicate and the iteration ordering of each data structure used in this lab, including your temporary variables. The GraphvizToF converter requires that the edges are printed according to a post-order traversal of the AST, so that it can reconstruct the AST bottom-up.

7.5 Non-determinism: Angelic, Demonic, and Arbitrary†

We previously saw non-determinism when studying finite automata. In that case we focused on *angelic* non-determinism: the automata would 'magically' (non-deterministically) choose the right path that will lead to success (acceptance).

In theory they also consider *demonic* non-determinism, where the machine always chooses a path that leads to the worst possible outcome.

In this lab we're seeing *arbitrary* non-determinism: the machine choose some path arbitrarily, and that path may lead to a good result or a bad result.

As engineers, we want to avoid non-determinism. It makes testing unnecessarily difficult. We transform non-deterministic machines into deterministic ones. We are careful to avoid unnecessary sources of non-determinism in our programs, such as iterating over hash structures or unregulated concurrency.

7.6 Translate one formula at a time

Flesh out the skeleton synthesizer in the provided TechnologyMapper class. The TechnologyMapper extends one of the Visitor classes, and in its visit methods it will create edges from the child nodes in the AST to their parents.

The TechnologyMapper class contains a field called substitutions that maps FExprs to FExprs. For now populate this data structure by mapping every FExpr AST node to itself. Populating this data structure in a more sophisticated manner is the next task.

7.7 Perform common subexpression elimination

An $\mathcal{F}$ program may contain common subexpressions. A smart synthesizer will synthesize shared hardware for these common subexpressions. Consider the following simple $\mathcal{F}$ program where the subexpression A or B appears twice:

```
X <= D and (A or B);
Y <= E and (A or B);
```

We do not have to worry about variables changing values in $\mathcal{F}$ because all expressions in $\mathcal{F}$ are *referentially transparent*: *i.e.*, they always have the same value no matter where they occur in the program. Referential transparency is a common property of *functional* programming languages such as Scheme, Lisp, Haskell, ML, and $\mathcal{F}$. Consider the following program fragment written in an *imperative* language:

```
a = 1;
b = 2;
x = a + b; // = 3
a = 3;
y = a + b; // = 5
```

The subexpression a + b occurs twice in this fragment but cannot be eliminated because the value of a has changed. A *dataflow analysis* is required to determine if the values of the variables have changed.

Figure 7.15 shows the circuit produced by a synthesizer that does not perform common subexpression elimination: there are two OR gates that both compute A or B. A more sophisticated synthesizer that does perform common subexpression elimination would synthesize a circuit like the one in Figure 7.16 in which there is only a single OR gate. It is also possible to have common subexpressions within a single formula, for example: Z <= (A or B) and !(A or B);

One way to write a common subexpression eliminator for $\mathcal{F}$ is as follows:

a. Build up a table of FExpr substitutions. Suppose our $\mathcal{F}$ program has three occurrences of A or B: $(\text{A or B})_1$, $(\text{A or B})_2$, and $(\text{A or B})_3$. These three expressions are in the same *equivalence class* according to our isomorphic() method. We select one representative from this group, say $(\text{A or B})_2$, and use that to represent all of the isomorphic expressions. So our substitution table will contain the tuples $\langle(\text{A or B})_1, (\text{A or B})_2\rangle$, $\langle(\text{A or B})_2, (\text{A or B})_2\rangle$, $\langle(\text{A or B})_3, (\text{A or B})_2\rangle$. In other words, we'll use $(\text{A or B})_2$ in place of $(\text{A or B})_1$ and $(\text{A or B})_3$.

   This substitution table can be built in the following way. First, compare every FExpr in the program with every other FExpr with the isomorphic() method to discover the equivalence classes. For each equivalence class pick a representative (doesn't matter which one), and then make an entry in the substitution table mapping each FExpr in the equivalence class to the representative.

Why do we use the isomorphic() method here instead of the equivalent() method? Because the isomorphic() method runs in polynomial time whereas the equivalent method runs in exponential time. Our common subexpression elimination only makes engineering sense if it costs polynomial time. If we were to spend exponential time then we could do more sophisticated circuit minimization, such as computing the prime implicants with the Quine-McCluskey algorithm.

b. Visit the AST and produce edges from children FExprs to parent FExprs using the substitution table.

See the provided utility methods in the skeleton TechnologyMapper class.

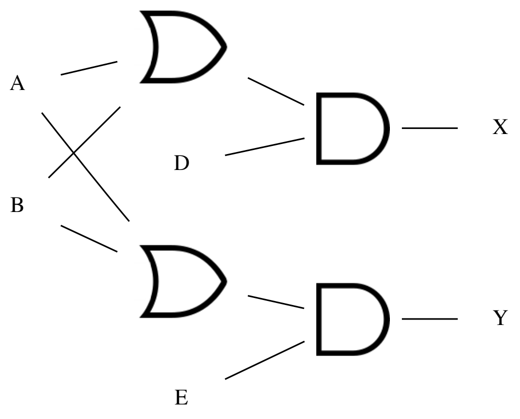


Figure 7.15: Synthesized circuit *without* common subexpression elimination for $\mathcal{F}$ program X <= D and (A or B); Y <= E and (A or B). There are two gates synthesized that both compute A or B.

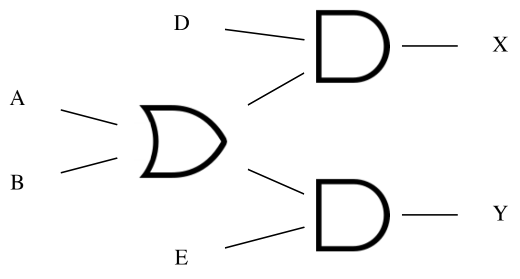


Figure 7.16: Synthesized circuit *with* common subexpression elimination for $\mathcal{F}$ program X <= D and (A or B); Y <= E and (A or B). There is only one gate that computes the value A or B, and the output of this gate is fed to two places.

7.8 *Evaluation*

The last pushed commit before the deadline is evaluated both on the shared test inputs and on a set of secret test inputs according to the weights in the table below.

The first testing equation for this lab is as follows. Let f be the AST produced by your $\mathcal{F}$ parser.

f.equivalent(GraphVizToF(TechnologyMapper(simplify(f))))

In other words, take the output of your $\mathcal{F}$ parser, simplify it, convert it to dot (this is the part that you are doing, the technology mapper), then convert that output back to an $\mathcal{F}$ program (we provide this code for you), and compare this result to the original AST produced by the parser.

The second testing equation for this lab compares your output with the staff output (dot file). Let s name the staff dot file, and f names the AST produced by your $\mathcal{F}$ parser.

simplify(GraphVizToF(s)).equivalent(GraphVizToF(TechnologyMapper(simplify(f))))

The two equations above just check that you produced an equivalent circuit, but they do not measure the number of gates used in that circuit. A third component of the evaluation is comparing the number of gates in your circuit versus the staff circuit.

TestTechnologyMapper	Shared	Secret
correctness	50	20
gate count	20	10

The baseline gate count marks are 16/20 and 8/10. If you produce fewer gates than the staff solution you bump up to 20/20 and 10/10. If you produce more gates than the staff solution than for each extra gate we subtract one point from the baseline. Some strategies that might produce fewer gates but probably take too much time to implement include:

- Using DeMorgan's Laws or other algebraic identities.
- Implementing the Quine-McCluskey or ESPRESSO algorithms to compute the minimal form of the circuit (a much more advanced version of our simplifier).
- Translating our FExprs to Binary Decision Diagrams (BDDs).[8] BDDS are used in practice for functional technology mapping and circuit equivalence checking.

The Quine-McCluskey algorithm is not used in practice because it is exponential/NP-complete, which means it is too expensive for most real circuits. We can afford the price for the small circuits considered in this course. The ESPRESSO algorithm is efficient enough to be used in practice but is not guaranteed to find the global minimum. http://en.wikipedia.org/wiki/Espresso_heuristic_logic_minimizer http://en.wikipedia.org/wiki/Quine-McCluskey_algorithm

[8] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, Aug. 1986

MORAL: Reducing the gate count is an NP-complete problem. Our simplifier and common subexpression eliminator are both polynomial. No combination of polynomial algorithms is ever going to be a perfect solution to an NP-complete problem. The ESPRESSO algorithm is an example of a very good polynomial approximation for this NP-complete problem.

Compiler Concepts: program generation, name capture
Programming Concepts:

Lab 8
Simulation: $\mathcal{F} \rightarrow$ Java

Files:
ece351.f.SimulatorGenerator
Tests:
ece351.f.TestSimulatorGenerator

Consider the following *F* program: `X <= A or B;`. A simulator for this $\mathcal{F}$ program would read an input $\mathcal{W}$ file describing the values of A and B at each time step, and would write an output $\mathcal{W}$ file describing the values of X at each time step. Such a simulator is listed in Figure 8.1. Your task is to write a program that, given an input $\mathcal{F}$ program, will produce a Java program like the one in Figure 8.1. In other words, you are going to write a simulator generator. The generated simulators will perform the following steps:

a. Read the input $\mathcal{W}$ program.
b. Allocate storage for the output $\mathcal{W}$ program.
c. Loop over the time steps and compute the outputs.
d. Write the output $\mathcal{W}$ program.

An alternative to generating a simulator would be to write an $\mathcal{F}$ *interpreter*. What we are doing here is a writing an $\mathcal{F}$ *compiler*. We call it a 'simulator generator' because the term 'compile' is not clear in the context of $\mathcal{F}$: in the last lab we 'compiled' $\mathcal{F}$ to a circuit; here we 'compile' it to a simulator.

An *interpreter* evaluates a program with an input, but does not transform the program to another language. A compiler, by contrast, is almost the opposite: it translates a program to another language, but does not evaluate that program on any input.

It is usually the case that it takes less effort to write an interpreter, but that the target program's runtime is faster if it is compiled first.

The generated simulator will have a method to compute the value of each output pin. In our example $\mathcal{F}$ program listed above the output pin is `X`, and so the generated simulator in Figure 8.1 has a method named `X`. The body of an output pin method is generated by performing a *pre-order* traversal of the corresponding $\mathcal{F}$ expression AST. $\mathcal{F}$ expressions are written with operators *in-order*: that is, the operators appear between the variables. For example, in the $\mathcal{F}$ program we have the in-order expression `A + B`, while in the Java translation we have the pre-order expression `or(A, B)`.

8.1 Name Collision/Capture†

When we generate code in another language we need to be careful that the variable names we generate based on our input program are legal in the target language, and that they do not collide/capture an already existing name. All legal $\mathcal{W}$ and $\mathcal{F}$ variable names are also legal Java variable names, so we don't need to worry about the first point here. On the second point, notice that the generated Java variable names in Figure 8.1 are prefixed with '`in_`' or '`out_`', and that none of the boilerplate names have such prefixes.

A name capture problem occurred in some of the utility code. The FProgram.equivalent() method checks that the equivalence of two $\mathcal{F}$ programs by translating them to SAT. For convenience, that process first translates the $\mathcal{F}$ program to Alloy, and then the Alloy is translated to SAT. An older version of this translation did not take care to avoid name capture, so if the input $\mathcal{F}$ program had variables corresponding to Alloy keywords (*e.g.*, 'this', 'int') then the Alloy to SAT translation would fail.

Figure 8.1: Simulator for $\mathcal{F}$ program x <= a or b;

```
import java.util.*;
import ece351.w.ast.*;
import ece351.w.parboiled.*;
import static ece351.util.Boolean351.*;
import ece351.util.CommandLine;
import java.io.File;
import java.io.FileWriter;
import java.io.StringWriter;
import java.io.PrintWriter;
import java.io.IOException;
import ece351.util.Debug;

public final class Simulator_ex11 {
    public static void main(final String[] args) {
        final String s = File.separator;
        // read the input F program
        // write the output
        // read input WProgram
        final CommandLine cmd = new CommandLine(args);
        final String input = cmd.readInputSpec();
        final WProgram wprogram = WParboiledParser.parse(input);
        // construct storage for output
        final Map<String,StringBuilder> output = new LinkedHashMap<String,StringBuilder>();
        output.put("x", new StringBuilder());
        // loop over each time step
        final int timeCount = wprogram.timeCount();
        for (int time = 0; time < timeCount; time++) {
            // values of input variables at this time step
            final boolean in_a = wprogram.valueAtTime("a", time);
            final boolean in_b = wprogram.valueAtTime("b", time);
            // values of output variables at this time step
            final String out_x = x(in_a, in_b) ? "1 " : "0 ";
            // store outputs
            output.get("x").append(out_x);
        }
        try {
            final File f = cmd.getOutputFile();
            f.getParentFile().mkdirs();
            final PrintWriter pw = new PrintWriter(new FileWriter(f));
            // write the input
            System.out.println(wprogram.toString());
            pw.println(wprogram.toString());
            // write the output
            System.out.println(f.getAbsolutePath());
            for (final Map.Entry<String,StringBuilder> e : output.entrySet()) {
                System.out.println(e.getKey() + ":" + e.getValue().toString()+ ";");
                pw.write(e.getKey() + ":" + e.getValue().toString()+ ";\n");
            }
            pw.close();
        } catch (final IOException e) {
            Debug.barf(e.getMessage());
        }
    }
    // methods to compute values for output pins
    public static boolean x(final boolean a, final boolean b) { return or(a, b) ; }
}
```

Figure 8.2 lists a visitor that determines all of the input variables used by an $\mathcal{F}$ expression AST. This code is listed here for your reference. It is also in your repository. You will need to call this code when writing your simulator generator.

Figure 8.2: Implementation of DetermineInputVars

```
import ece351.common.ast.Expr;
import ece351.common.ast.NAndExpr;
import ece351.common.ast.NOrExpr;
import ece351.common.ast.NaryAndExpr;
import ece351.common.ast.NaryOrExpr;
import ece351.common.ast.NotExpr;
import ece351.common.ast.OrExpr;
import ece351.common.ast.VarExpr;
import ece351.common.ast.XNOrExpr;
import ece351.common.ast.XOrExpr;
import ece351.common.visitor.PostOrderExprVisitor;
import ece351.f.ast.FProgram;


public final class DetermineInputVars extends PostOrderExprVisitor {
    private final Set<String> inputVars = new LinkedHashSet<String>();
    private DetermineInputVars(final AssignmentStatement f) { traverseExpr(f.expr); }
    public static Set<String> inputVars(final AssignmentStatement f) {
        final DetermineInputVars div = new DetermineInputVars(f);
        return Collections.unmodifiableSet(div.inputVars);
    }
    public static Set<String> inputVars(final FProgram p) {
        final Set<String> vars = new TreeSet<String>();
        for (final AssignmentStatement f : p.formulas) {
            vars.addAll(DetermineInputVars.inputVars(f));
```

8.2 *Evaluation*

The last pushed commit before the deadline is evaluated both on the shared test inputs and on a set of secret test inputs according to the weights in the table below. Let 'f' name the $\mathcal{F}$ program we want to simulate, let 'w' name an input $\mathcal{W}$ program for it, and let 'wStaff' name the output of the staff's simulator for this $\mathcal{F}/\mathcal{W}$ combination. The testing equation is:

```
gen(f).simulate(w).isomorphic(wStaff)
```

Can we compare your generated Java files directly against the staff generated Java files? No, because Java is a Turing Complete language, we know from the Halting Problem that this is undecidable in general. So instead we compare the generated Java files indirectly, by observing if they compute the same outputs for the same inputs. In this case the inputs and output are $\mathcal{W}$ programs, and since $\mathcal{W}$ is a regular language we can easily compare those.

	Current	New
TestSimulatorGenerator	70	30

8.3 *Bonus Lab: $\mathcal{F}$ to* x86

This is a complete bonus lab, with the same weight as all of the other labs. But it will probably take you four times longer than any of the other labs, so the return on investment of time is poor. There is no official instructional support for this lab beyond what is written here. It is not for the faint of heart. It is expected that less than 5% of the class will attempt this bonus lab.

Objective: Produce x86 assembler for the methods that compute the values for the output pins, and link it to the Java main method via the jni (Java Native Interface).

The main method will still be in Java, and you will still use your Java $\mathcal{W}$ parser. It's just the methods that compute the values for the output pins that are being ported to assembler.

Recommended first step: First produce C code for the output pin methods and link that to the Java code via jni. This first step will get you familiar with how to integrate the parts, so you can isolate and focus on the assembler issues in the next step.

Horseshoes and hand grenades. There are no part marks. Doing the first step gets you nothing but skills and knowledge. Do not start this exercise if you are not committed to investing a substantial amount of time in it. Do it because you want the skills and the knowledge and the badge of honour: there are more efficient ways to earn extra marks at school.

Evaluation is by personal interview with the instructor, any time before the last class (the bonus does not have to be completed at the same time as the mandatory lab 8 work).

Compiler Concepts:
Programming Concepts:

Lab 9
VHDL *Recognizer & Parser*

Files:
ece351.vhdl.VRecognizer.java
ece351.vhdl.VParser.java

Tests:
ece351.vhdl.test.TestVRecognizerAccept
ece351.vhdl.test.TestVRecognizerReject
ece351.vhdl.test.TestVParser

In this lab we will build a recognizer and a parser using Parboiled for a very small and simple subset of VHDL. The grammar for this subset of VHDL is listed in Figure 9.1. Restrictions on our subset of VHDL include:

- only bit (boolean) variables
- no stdlogic
- no nested ifs
- no aggregate assignment
- no combinational loops
- no arrays
- no generate loops
- no case
- no when (switch)
- inside process: either multiple assignment statements or multiple if statements; inside an if there can be multiple assignment statements
- no wait
- no timing
- no postponed
- no malformed syntax (or, no good error messages)
- identifiers are case sensitive
- one architecture per entity, and that single architecture must occur immediately after the entity

In W2013 the median time to complete this lab was about 9 hours. Therefore, we have made three changes to this lab for S2013 to hopefully bring this time down to the five hour budget.

First, we added `append` methods to the VHDL AST classes (as we also did for $\mathcal{F}$ this term). This means that your parser does not need to place ImmutableList objects on the stack directly, which many students found difficult to manage (and certainly the syntax for the casting is complex). This also means that you can construct your AST in a top-down manner, which many students seem to find easier. Both $\mathcal{W}$ and $\mathcal{F}$ were changed to this style this term as well, so this is what you are already familiar with.

Second, we provided implementations of the object contract methods on the VHDL AST classes for you, instead of requiring you to fill them in.

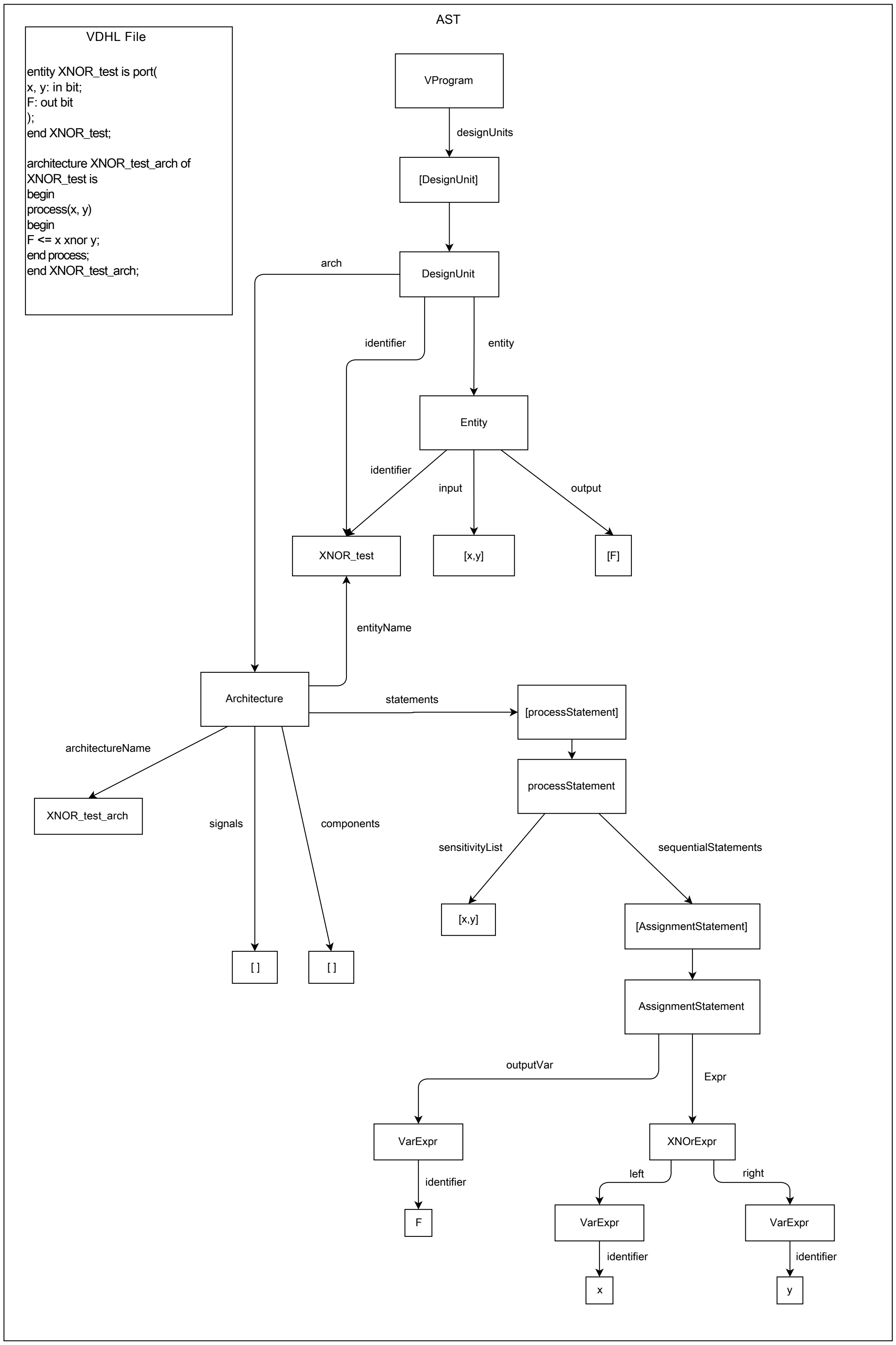
Third, we removed the desugarer and define before use parts of this lab. So now it's just the parser.

Program	→	(*DesignUnit*)*
DesignUnit	→	*EntityDecl ArchBody*
EntityDecl	→	**‘entity’** *Id* **‘is’** **‘port’** **‘(’** *IdList* **‘:’** **‘in’** **‘bit’** **‘;’** *IdList* **‘:’** **‘out’** **‘bit’** **‘)’** **‘;’** **‘end’** (**‘entity’** | *Id*) **‘;’**
IdList	→	*Id* (**‘,’** *Id*)*
ArchBody	→	**‘architecture’** *Id* **‘of’** *Id* **‘is’** (**‘signal’** *IdList* **‘:’** **‘bit’** **‘;’**)? **‘begin’** (*CompInst*)* (*ProcessStmts* | *SigAssnStmts*) **‘end’** *Id* **‘;’**
SigAssnStmts	→	(*SigAssnStmt*)$^+$
SigAssnStmt	→	*Id* **‘<=’** *Expr* **‘;’**
ProcessStmts	→	(*ProcessStmt*)$^+$
ProcessStmt	→	**‘process’** **‘(’** *IdList* **‘)’** **‘begin’** (*IfElseStmts* | *SigAssnStmts*) **‘end’** **‘process’** **‘;’**
IfElseStmts	→	(*IfElseStmt*)$^+$
IfElseStmt	→	**‘if’** *Expr* **‘then’** *SigAssnStmts* **‘else’** *SigAssnStmts* **‘end’** **‘if’** (*Id*)? **‘;’**
CompInst	→	*Id* **‘:’** **‘entity’** **‘work.’** *Id* **‘port’** **‘map’** **‘(’** *IdList* **‘)’** **‘;’**
Expr	→	*XnorTerm* (**‘xnor’** *XnorTerm*)*
XnorTerm	→	*XorTerm* (**‘xor’** *XorTerm*)*
XorTerm	→	*NorTerm* (**‘nor’** *NorTerm*)*
NorTerm	→	*NandTerm* (**‘nand’** *NandTerm*)*
NandTerm	→	*OrTerm* (**‘or’** *OrTerm*)*
OrTerm	→	*AndTerm* (**‘and’** *AndTerm*)*
AndTerm	→	*EqTerm* (**‘=’** *EqTerm*)*
EqTerm	→	**‘not’** *EqTerm* | **‘(’** Expr **‘)’** | *Var* | *Constant*
Constant	→	**‘0’** | **‘1’**
Var	→	*id*
Id	→	*Char* (*Char* | *Digit* | **‘_’**)*
Char	→	`[A-Za-z]`
Digit	→	`[0-9]`

Figure 9.1: Grammar for VHDL.

This grammar has gone through a number of evolutions. First, David Gao and Rui Kong (3B) downloaded a complete VHDL grammar implementation for the ANTLR tool. They found that grammar was too complicated to use, so they started writing their own with ANTLR. Then Alex Duanmu and Luke Li (1B) reimplemented their grammar in TXL and simplified it. Aman Muthrej (3A) reimplemented it in Parboiled and simplified it further. Wallace Wu (TA) refactored Aman's code. Michael Thiessen (3A) helped make the expression sub-grammar more consistent with the grammar of $\mathcal{F}$.

There will no doubt be improvements and simplifications that you think of, and we would be happy to incorporate them into the lab.



```
entity XNOR_test is port(
x, y: in bit;
F: out bit
);
end XNOR_test;

architecture XNOR_test_arch of
XNOR_test is
begin
process(x, y)
begin
F <= x xnor y;
end process;
end XNOR_test_arch;
```

Figure 9.2: Object diagram for example vhdl ast

9.1 *Keywords and Whitespaces*

To make the code for the recognizer and parser a little bit more legible (and easier to implement), classes `VRecognizer` and `VParser` inherit a common base class called `VBase`,[1] which contains a set of rules that will match keywords found in the language in a case-independent manner (since VHDL is case-insensitive[2] – *e.g.*, both 'signal' and 'SIGNAL' represent the same keyword in the language).

[1] `VBase` itself extends BaseParser351, which provides extra utility and debugging methods over Parboiled's BaseParser class.

[2] In some VHDL compilers, there may be some exceptions, where the case sensitivity matters a handful of grammar productions

Figure 9.3 is an example of one of the rules found in `VBase`. Here, the rule matches the keyword 'signal' and as there is no whitespace handling, your code should ensure that at least one whitespace exists between this keyword and the next token/string that is matched.

```
return SingleKeyword(NOR);
```

Figure 9.3: Implementation of rule used to match the keyword 'signal'

9.2 VHDL *Recognizer*

Sources:
ece351.vhdl.VRecognizer
Tests:
ece351.vhdl.test.TestVRecognizerAccept
ece351.vhdl.test.TestVRecognizerReject

Write a recognizer using Parboiled for the VHDL grammar defined in Figure 9.1. Follow the methodology that we have been developing this term:

- Make a method for every non-terminal [Lab 1.2]
- Convert EBNF multiplicities in the grammar into Parboiled library calls [Lab 5.1]
- Convert the lexical specification into Parboiled library calls [Lab 5.1]
- Remember to explicitly check for EOI

You should be able to copy and paste the code from your $\mathcal{F}$ parboiled recognizer with minimal modifications, since $\mathcal{F}$ is a subset of our VHDL grammar.

9.3 VHDL *Parser*

Sources:
ece351.vhdl.VParser
Libraries:
ece351.vhdl.ast.*
ece351.common.ast.*
Tests:
ece351.vhdl.test.TestVParser

Write a parser using Parboiled for the VHDL grammar defined in Figure 9.1. You do not need to write a pretty-printer nor the object contract methods: these are provided for you in the new VHDL AST classes, and you already wrote them in the shared AST classes (ece351.common.ast). You do not need to edit the AST classes. Just write the parser. Follow the steps we learned earlier in the term:

- Draw out the evolution of the stack for a simple input [Figure 5.3].
- Print out our VHDL grammar.
- Annotate the printed copy of the grammar with where stack manipulations occur according to your diagram.

- Calculate the maximum and expected size of the stack for different points in the grammar.
- Copy your recognizer code into the parser class.
- Add some (but not all) actions (push, pop, *etc.*).
- Add some checkType() and debugmsg() calls [Figure 5.5].
- Run the TestVParser harness to see that it passes.
- Add more actions, checkType() calls, *etc.*.
- Repeat until complete.

When you implement the vhdl parser, there may be a few grammar productions shown in Figure 9.1 where you will need to rewrite each of these rules in a different way so that you can instantiate all of the required objects used to construct the AST of the program being parsed.

9.4 *Engineering a Grammar*†

For simple languages like $\mathcal{W}$ and $\mathcal{F}$ we can write a 'perfect' recognizer: that is, a recognizer that accepts all grammatical strings and rejects all ungrammatical strings. For more complicated languages, such as vhdl, it is common to design a grammar for a particular task, and that task may be something other than being an ideal recognizer. For example, the grammar might accept some strings that should be rejected.

Industrial programming environments such as Eclipse often include multiple parsers for the same language. Eclipse's Java development tools include three parsers: one that gives feedback on code while the programmer is editing it (and so the code is probably ungrammatical); one that gives feedback on the code when the programmer saves the file (when the code is expected to be grammatical); and one that is part of the compiler. As you may surmise from this example, the task of giving good feedback to a programmer about ungrammatical code can be quite different from the task of translating grammatical code.

The TestVRecognizerReject harness tests for some obvious cases that should be rejected. Do not invest substantial effort into trying to reject more cases.

9.5 *Evaluation*

The last pushed commit before the deadline is evaluated both on the shared test inputs and on a set of secret test inputs according to the weights in the table below.

	Current	New
TestVParboiledRecognizerAccept	20	5
TestVParboiledRecognizerReject	5	0
TestVParboiledParser	45	25

You do not get any marks for any other parser tests until TestObjectContractF passes everything. You do not get any marks for rejection tests until some acceptance tests pass.

TestVParboiledRecognizer just runs the recognizer to see if it crashes. TestVParboiledParser checks the following equation:

$\forall$ v : *.vhd | parse(v).isomorphic(parse(prettyprint(parse(v))))

Compiler Concepts: desugaring, function inlining
Programming Concepts:

Lab 10

VHDL → VHDL: *Desugaring & Elaboration*

In this lab, you will be writing two VHDL to VHDL transformers using the Visitor design pattern. The first, and simpler one, will simply rewrite expressions to replace operators like XOR with their equivalent formulation in terms of AND, OR, and NOT. We call this *desugaring*, where operators like XOR are considered to be *syntactic sugar*: things that might be convenient for the user, but do not enlarge the set of computations that can be described by the language.

The second, and more sophisticated, transformer will expand and inline any component instance declared within an architecture of a design unit if the component is defined within the same VHDL source file. The procedure that this transformer performs is known as *elaboration*. Elaboration is essential for us to process the four bit ripple-carry adder, for example, because it comprises four one-bit full adders.

10.1 Pointers, Immutability and Tree Rewriting with Visitor†

Go back and reread §1.5. Go to PythonTutor.com and understand the visualizations that show how variables point to objects. You really need to understand pointers, aliasing, and immutability to work on this lab. The TA's report that although these are fundamental concepts we reviewed at the beginning of term, many students are still unclear on them at this point.

Immutable means that an object doesn't change. But this whole lab is, in some sense, about changing our VHDL AST: first by rewriting exotic operators like XOR in terms of the basic (AND, OR, NOT) operators (desugaring); and then by inlining the definitions of subcomponents (elaboration). How can we change the AST if it's immutable?

The same we change any immutable object: we don't. We create a new AST that is like the old AST except for the parts that we want to be different.

TODO: draw pictures of tree rewrite of `X <= A or B and '1';`

10.2 VHDL → VHDL: *Desugaring*

Sources:
ece351.vhdl.DeSugarer
Tests:
ece351.vhdl.tests.TestDeSugarer

Often times, languages provide redundant syntactic constructs for the purpose of making code easier to write, read, and maintain for the programmer. These redundant constructs provide 'syntactic sugar'. In this part of the lab, you will transform VHDL programs that you parse into valid VHDL source code that 'desugars' the expressions in the signal assignment statements. In other words, the desugaring process reduces the source code into a smaller set of (i.e., more basic) constructs.

In VHDL, desugaring may be useful in cases where the types of available logic gates are limited by the programmable hardware you might be using to run your VHDL code. For example, if the programmable hardware only comprises of NAND gates, a VHDL synthesizer will be required to rewrite all of the logical expressions in your code using NAND operators.

For this part of the lab, write a 'desugarer' that traverses through an AST corresponding to an input VHDL program and rewrites all expressions so that all expressions in the *transformed* AST only consist of AND, OR, and NOT logical operators. This means that expressions containing XOR, NAND, NOR, XNOR, and = operators must be rewritten. For example, the expression $x \oplus y$ (XOR) is equivalent to:

$$x \oplus y \equiv \; x\cdot !y + !x \cdot y \tag{10.1}$$

Table 10.1 is the truth table for the '=' operator: By observation, we can see that this truth table is equivalent to that of XNOR.

x	y	$x = y$
0	0	1
0	1	0
1	0	0
1	1	1

Table 10.1: Truth table for the '=' operator. Equivalent to XNOR.

10.3 *Elaboration*

Sources:
ece351.vhdl.Elaborator
Tests:
ece351.vhdl.test.TestElaborator

The following sections describe the expected behaviour of the elaborator.

10.3.1 *Inlining Components without Signal List in Architecture*

Consider the VHDL source shown in Figure 10.1. Here, we have two design units, `OR_gate_2` and `four_port_OR_gate_2`, where the architecture body corresponding to `four_port_OR_gate_2` instantiates two instances of `OR_gate_2`, namely `OR1` and `OR2` (lines 20 and 21).

When the elaborator processes this program, it will check the design units sequentially. In this example, `OR_gate_2` is checked first. The architecture body corresponding to `OR_gate_2` does not instantiate any components, so the elaborator does not do anything to this design unit and moves onto the next design unit. In the architecture body, `four_port_structure`, we see that there are two components that are instantiated. Since there are components within this architecture body, the elaborator should then proceed to inline the behaviour of the components into `four_port_structure` and make the appropriate parameter/signal substitutions.

Figure 10.1: VHDL program used to illustrate elaboration.

```
 1  entity OR_gate_2 is port (
 2      x , y: in bit;
 3      F: out bit
 4  );
 5  end OR_gate_2;
 6
 7  architecture OR_gate_arch of OR_gate_2 is begin
 8      F <= x or y;
 9  end OR_gate_arch;
10
11  entity four_port_OR_gate_2 is port (
12      a,b,c,d : in bit;
13      result : out bit
14  );
15  end four_port_OR_gate_2;
16
17  architecture four_port_structure of four_port_OR_gate_2 is
18      signal e, f : bit;
19  begin
20      OR1: entity work.OR_gate_2 port map(a,b,e);
21      OR2: entity work.OR_gate_2 port map(c,d,f);
22      result <= e or f;
23  end four_port_structure;
```

Consider the component `OR1`. The elaborator will search through the list of design units that make up the program and determine the design unit that the component is instantiating. In this example, `OR1` is an instance of the design unit `OR_gate_2` (see the string immediately following "work." on line 20). Then the elaborator proceeds to determine how the signals used in the port map relate to the ports defined in the entity declaration of `OR_gate_2`. `OR1` maps the signals `a`, `b`, and `e`, to the ports `x`, `y`, and `F` of the entity `OR_gate_2`, respectively. Using the private member, `current_map`, in the Elaborator class will help you with the signal substitutions that occurs when a component is inlined. After the mapping is established, the elaborator then proceeds to replace `OR1` by inlining the architecture body corresponding to the entity `OR_gate_2` into `four_port_structure`.

The same procedure is carried out for the component `OR2` and we now will have an equivalent architecture body as shown in Figure 10.2. Lines 5 and 6 in Figure 10.2 corresponds to the inlining of `OR1` and `OR2` (found in lines 20 and 21 from Figure 10.1).

```
1  begin
2          result <= ( e or f );
3          e <= ( a or b );
4          f <= ( c or d );
5  end four_port_structure;
6
7  entity eight_port_OR_gate_2 is port(
8          x0, x1, x2, x3, x4, x5, x6, x7 : in bit;
```

Figure 10.2: Elaborated architecture body, `four_port_structure`.

10.3.2 Inlining Components with Signal List in Architecture

In addition to substituting the input and output pins for a port map, you will also encounter situations where there are signals declared in the architecture body that you are trying to inline to the design unit the elaborator is currently processing.

For example, consider VHDL source in Figure 10.3, which is the extension of the program in Figure 10.1.

The two components in `eight_port_structure` are instances of `four_port_OR_gate_2`; the architecture of `four_port_OR_gate_2` defines signals `e` and `f`. Now, if we elaborate, say, `OR1`, we determine the mapping as before for the input and output pins, but we also need to consider the signals defined within the architecture. If we simply add `e` and `f` to the list of signals of `eight_port_structure`, we will run into problems of multiply defined signals when we elaborate `OR2`; we will obtain a signal list with two `e`'s and two `f`'s. Furthermore, we will change the logical behaviour defined in `eight_port_structure`.

To address this issue, all internal signals that are added as a result of elaboration will be prefixed with '`comp<num>_`', where `<num>` is a unique identifier[1] used to ensure that the elaboration does not change the logical behaviour of the program. The result of elaborating `eight_port_OR_gate_2` is shown Figure 10.4.

[1] This number starts at 1 and increments for each component that is instantiated in the VHDL program. `<num>` is never reset.

Figure 10.3: Extension of the VHDL program shown in Figure 10.1.

```
entity eight_port_OR_gate_2 is port (
    x0, x1, x2, x3, x4, x5, x6, x7 : in bit;
    y : out bit
);
end eight_port_OR_gate_2;

architecture eight_port_structure of eight_port_OR_gate_2 is
    signal result1, result2 : bit;
begin
    OR1: entity work.four_port_OR_gate_2 port map(x0, x1, x2, x3, result1);
    OR2: entity work.four_port_OR_gate_2 port map(x4, x5, x6, x7, result2);
    y <= result1 or result2;
end eight_port_structure;
```

Figure 10.4: Elaborated architecture body, `eight_port_structure`.

```
entity eight_port_OR_gate_2 is port(
        x0, x1, x2, x3, x4, x5, x6, x7 : in bit;
        y : out bit
);
end eight_port_OR_gate_2;
architecture eight_port_structure of eight_port_OR_gate_2 is
        signal result1, result2, comp3_e, comp3_f, comp4_e, comp4_f : bit;
begin
        y <= ( result1 or result2 );
        result1 <= ( comp3_e or comp3_f );
        comp3_e <= ( x0 or x1 );
        comp3_f <= ( x2 or x3 );
        result2 <= ( comp4_e or comp4_f );
        comp4_e <= ( x4 or x5 );
        comp4_f <= ( x6 or x7 );
end eight_port_structure;
```

10.3.3 *Inlining Components with Processes in Architecture*

The previous examples demonstrated the behaviour of inlining components with architectures that only contain signal assignment statements. When the elaborator encounters processes in the inlining, a similar procedure is performed. The main difference in the procedure for processes is to make the appropriate signal substitutions in sensitivity lists and if-else statement conditions.

10.3.4 *Notes*

- The elaborator will only expand components when its corresponding design unit is also defined in the same file.
- The elaborator processes the design units in sequential order. We assume that the VHDL programs we are transforming are written so that you do not encounter cases where the architecture that you are inlining contains components that have not yet been elaborated.
- We will assume that the VHDL programs being elaborated will not result in architecture bodies with a mixture of parallel signal assignment statements and process statements (so that the parser from Lab 9 can parse the transformed programs).

10.4 *Evaluation*

The last pushed commit before the deadline is evaluated both on the shared test inputs and on a set of secret test inputs according to the weights in the table below. The testing equation for desugaring is:

```
parse(v).desugar().isomorphic(parse(vStaff))
```

Similarly, the testing equation for elaboration is:

```
parse(v).elaborate().isomorphic(parse(vStaff))
```

	Shared	Secret
TestDeSugarer	20	10
TestElaborator	50	20

Choice: either do this lab or do the lab marked as LAB12. Do not do both.

Lab 11 VHDL *Process Splitting & Combinational Synthesis*

Files:
ece351.vhdl.Splitter.java
ece351.vhdl.Synthesizer.java
Tests:
ece351.vhdl.test.TestSplitter
ece351.vhdl.test.TestSynthesizer

For this lab, you will be writing code to perform two other transformations on VHDL programs. The first transformation is a VHDL to VHDL transformation, which we call *process splitting*. Process splitting involves breaking up if/else statements where multiple signals are being assigned new values in the if and else clauses.

In the second part of this lab, you will be translating VHDL to $\mathcal{F}$, which we call *combinational synthesis*. The combinational synthesizer will take the VHDL program output from the process splitter and generate a valid $\mathcal{F}$ program from it.

11.1 Process Splitter

The process splitter will perform the necessary transformations to VHDL program ASTs so that exactly one signal is being assigned a value in a process. Consider the VHDL code shown in Figure 11.1.

Here, we have a process in the architecture `behv1` of the entity `Mux` the requires splitting because in the if and else clauses, there are two signals that are being assigned new values: `O0` and `O1`. The splitter should observe this and proceed to replace the current process with two new processes: one to handle the output signal `O0` and the other to handle `O1`. Figure 11.2 shows the splitter's output when the code in Figure 11.1 is processed. Note that the sensitivity lists for the two new processes only contain the set of signals that may cause a change to the output signals.

Figure 11.1: Example VHDL program used to illustrate process splitting.

```
entity Mux is port(
    I3,I2,I1,I0,S: in bit;
    O0,O1: out bit
);
end Mux;

architecture behv1 of Mux is
begin
  process(I3,I2,I1,I0,S)
    begin
        if (S = '0') then
            O0 <= I0;
            O1 <= I2;
        else
            O0 <= I1;
            O1 <= I3;
        end if;
    end process;

end behv1;
```

Figure 11.2: The resulting VHDL program of Figure 11.1 after process splitting.

```
entity Mux is port(
        I3, I2, I1, I0, S : in bit;
        O0, O1 : out bit
);
end Mux;
architecture behv1 of Mux is

begin
        process ( S, I0, I1 )
                begin
                        if ( ( not ( ( ( S and ( not ( '0' ) ) ) or ( ( not ( S ) ) and '0' ) ) ) ) ) then
                                O0 <= I0;
                        else
                                O0 <= I1;
                        end if;
                end process;
        process ( S, I2, I3 )
                begin
                        if ( ( not ( ( ( S and ( not ( '0' ) ) ) or ( ( not ( S ) ) and '0' ) ) ) ) ) then
                                O1 <= I2;
                        else
                                O1 <= I3;
                        end if;
                end process;
end behv1;
```

11.2 Splitter Notes

- Assume that there is exactly one assignment statement in the if body and one assignment statement in the else body that write to the same output signal. This implies that you do not need to handle the case where latches are inferred. Making this assumption should reduce the amount of code you need to write for this lab.
- The private variable `usedVarsInExpr` is used to store the variables/signals that are used in a VHDL expression. This is helpful when you are trying to create sensitivity lists for new processes.

11.3 Synthesizer

After parsing the input VHDL source file and performing all of the transformations (*i.e.*, desugaring, elaborating, and process splitting), the synthesizer will traverse the (transformed) AST (of the VHDL program), extract all expressions in the program, and generate an $\mathcal{F}$ program containing these expressions.

In §8 we generated a Java program by constructing a string. We noted that there was no guarantee that the string we generated would be a legal Java program. One way to ensure that we generate a legal program is to build up an AST instead of building up a string. Of course, to do that we need to have AST classes, which we do not have for Java — but do have for $\mathcal{F}$.

Because the $\mathcal{F}$ grammar and AST classes are a subset of those for VHDL, we can simply reuse many of the expressions from the input VHDL program with only minor modification to rename the variables to avoid name collision.

11.3.1 If/Else Statements

Whenever an if/else statement is encountered, first extract the condition and add it to the $\mathcal{F}$ program. Because there is no assignment that occurs in the condition expression, generate an output variable for the condition when you output the condition to the $\mathcal{F}$ program. The output variable will be of the form: '`condition<num>`', where `<num>` = 1, 2, 3, ... `<num>` is incremented each time you encounter a condition in the VHDL program and is never reset.

After appending the condition of the if/else statement to the $\mathcal{F}$ program, construct an assignment for the output variable like so:[1]

```
if ( vexpr ) then
    output <= vexpr1;
else
    output <= vexpr2;
end if;
```

[1] Note that process splitting is useful here because we will only have one signal assignment statement in the if- and else- bodies, where both statements assign the expressions to the same output signal.

The formulas that should be appended to the $\mathcal{F}$ program is:

```
condition<num> <= vexpr;
output <= ( condition<num> and (vexpr1) or (not condition<num>) and (vexpr2) );
```

Let's refer to the variable named conditionX as an *intermediate variable* because it is not intended as part of the circuit's final output, but is used in computing the final output pins. Our $\mathcal{F}$ simulator generator and technology mapper do not support these intermediate variables. Removing them is not hard: we just inline their definition wherever they are used. We will provide you with code to do that later.

The synthesizer here is not the only place where these intermediate variables are created. They also occur in elaboration. They are helpful for debugging, so it's worth keeping them here and removing them later.

11.3.2 Example Synthesizing Signal Assignment Statements

Consider the VHDL program shown in Figure 11.3. When the synthesizer processes this program[2], an $\mathcal{F}$ program consisting of a single formula (corresponding to) is generated (see Figure 11.4).

[2] after desugaring, elaborating, and splitting

Figure 11.3: Example used to illustrate synthesizing assignment statements.

```
entity NOR_gate is port(
    x,y: in bit;
    F: out bit
);
end NOR_gate;

architecture NOR_gate_arch of NOR_gate is
begin

    F <= x nor y;

end NOR_gate_arch;
```

Figure 11.4: Synthesized output of the program in Figure 11.3.

```
NOR_gateF <= ( not ( NOR_gatex or NOR_gatey ) );
```

11.3.3 Example Synthesizing If/Else Statements

Consider the VHDL program show in Figure 11.5. After performing all of the VHDL transformations that you have written, the synthesizer will generate the $\mathcal{F}$ program shown in Figure 11.6.

In Figure 11.6, observe that the synthesizer translates the if/else statement in Figure 11.5 into two $\mathcal{F}$ formulae. The first formula that is generated corresponds to the condition that is checked in the statement (i.e., `x='0' and y='0'`). The second formula combines the expressions found in the if and else bodies so that the expression in the if-body is assigned to `F` if the condition is true; otherwise, the expression in the else-body is assigned to `F`.

```
entity NOR_gate is port(
    x, y: in bit;
    F: out bit
);
end NOR_gate;

architecture NOR_gate_arch of NOR_gate is
begin
    process(x, y)
    begin
        if (x='0' and y='0') then
            F <= '1';
        else
            F <= '0';
        end if;
    end process;
end NOR_gate_arch;
```

Figure 11.5: Example used to illustrate synthesizing assignment statements.

```
condition1 <= ( ( not ( ( NOR_gatex and ( not '0' ) ) or ( ( not NOR_gatex )
    and '0' ) ) ) and ( not ( ( NOR_gatey and ( not '0' ) ) or ( ( not
    NOR_gatey ) and '0' ) ) ) );
NOR_gateF <= ( condition1 and ( '1' ) ) or ( ( not condition1 ) and ( '0' ) );
```

Figure 11.6: Synthesized output of the program in Figure 11.5.

11.4 *Evaluation*

The last pushed commit before the deadline is evaluated both on the shared test inputs and on a set of secret test inputs according to the weights in the table below.

	Current	New
TestSplitter	40	10
TestSynthesizer	40	10

Lab 12
Simulation: $\mathcal{F} \rightarrow$ Assembly

This lab by Nicholas Champion.

Choice: You can do either this lab or the original LAB11. Don't do both.
There is no official instructional support for this lab. Do not attempt this lab if you are not comfortable learning assembly on your own and configuring your system to work with the assembler that you have chosen.

In Lab 8 you generated Java code from an $\mathcal{F}$ program. That Java code would read and write $\mathcal{W}$ files, simulating the $\mathcal{F}$ program for each clock tick. In this lab you will make a variation of the simulator generator from Lab 8. The main loop is still the same, with the call to the $\mathcal{W}$ parser *etc.*. The only part that will be different is the methods you generate to compute the values of the output pins. For example, in Lab 8 you might have generated a method like the one shown in Figure 12.1. The point of this lab is to generate these methods in assembly, and have the main body (still in Java) call the assembly versions.

```
// methods to compute values for output pins
public static boolean x(final boolean a, final boolean b) { return or(a, b) ; }
```

Figure 12.1: Generate this method in assembly rather than in Java

12.1 Which assembler to use?

Options include:

- MASM32. Requires a 32 bit JDK. This is the option supported by the description below and the skeleton code. The other options have no instructional support.
- GCC with inline assembly.
- JWasm http://www.japheth.de/JWasm.html. JWasm is an open-source fork of the Watcom assembler (Wasm). It is written in C and is MASM compatible. The 'J' here apparently stands for the maintainer (Japheth), and not for Java.

  Watcom, as you know, is from the University of Waterloo. UW has a long and famous history with compilers, that you can read a little bit about at http://www.openwatcom.com/index.php/Project_History

- *MIPS Assembler and Runtime Simulator (MARS).* A Java implementation of the RISC machine described in the Hennessey & Patterson computer architecture textbook. http://courses.missouristate.edu/kenvollmar/mars/ Because this is a simulator written in Java, it might avoid a lot of the systems issues of going to x86 assembly. Also, it might be easier to generate MIPS assembly than x86 (RISC instruction sets are usually simpler than CISC ones).

There might be some bonus marks available if you develop a skeleton and some instructional support for an assembler such as MARS that has no system dependencies outside of a regular JVM.

12.2 MASM32 *Configuration*

- MASM32 requires Windows (any version should work). MASM32 includes an old version of Microsoft's assembler ml.exe. Newer version are included with releases of Visual Studio releases and support newer instruction.
- Install the 32-bit JDK and configure your Eclipse installation to use this JDK. JNI and Windows operating systems require that 64-bit programs use 64-bit dynamic libraries (*i.e.*, what MASM32 will generate). 64-bit programs cannot use 32-bit libraries.
- A possible multi-platform alternative to experiment with, which is NOT-supported by the skeleton or test harness, is JWASM.
- It can be downloaded from http://www.masm32.com/
- It is recommended that you install MASM32 to C:\masm32
- Add (<Install Path>\bin) to your system path environment variable (*i.e.*, C:\masm32\bin)
- To check if it was added to the path, open a new command window and type "ml.exe". It should output "Microsoft (R) Macro Assembler Version 6.14.8444...".

12.3 MASM32 *JNI and x86 Assembly*

The skeleton and test harness created for solving this lab using MASM32 are based on the Lab 8 and so approximately a third of the redacted skeleton code can be taken from a lab 8 solution. The main difference from lab 8 is the means in which the F-statements are evaluated. In lab 8, methods were created and defined in Java for each statement by walking the tree using pre-order. In lab 12, these methods are to be implemented in x86 assembly. It is not possible to include inline assembly in Java as Java is executed in the JVM. Therefore the x86 is assembled and linked into a shared library (dynamic link library (DLL) on Windows). This library must be loaded by the generated java code using the Java Native Interface (JNI). JNI allows native functions to be added to the class which loads the library. The library load in the skeleton is static while the methods are not static. Hence the main method will instantiate its own class and make calls

You can use the JavaH tool in order to see the function signatures. It is included with the JDK to generate the C header for a java class. You can modify TestSimulatorGeneratorMasm32.java to generate the headers files by uncommenting the lines which write the batch file. You will need to change the path to point to your JDK. To see how java call types map to standard C types (and hence ASM), see jni.h (in the JDK).

to its non-static native methods. JNI also imposes certain naming conventions on the signatures of functions declared in the shared library. For example the native method

```
public native boolean Simulator_opt1_or_true1 (final boolean a);
```

translates to the C shared library export signature

```
JNIEXPORT jboolean JNICALL Java_Simulator_1opt1_1or_1true1_Evalx(JNIEnv *, jobject, jboolean);
```

This C export translates to the x86 assembly signature (for the purposes of the this lab)

```
Java_Simulator_1opt1_1or_1true1_Evalx proc JNIEnv:DWORD, jobject:DWORD, in_a:BYTE
```

It is notable that JNI requires including pointers to the Java environment and the Java class instance. It is also notable that the underscores in the class name and the method name must be replaced with _1. Most of the function prototype details are already included in the skeleton.

Figure 12.2 is a simple example of the generate code for a shared library for a $\mathcal{F}$ Program.

Figure 12.2: Example assembly for an F Program

```
.386
.model flat,stdcall
option casemap:none

Java_Simulator_1opt1_1or_1true1_Evalx PROTO :DWORD, :DWORD, :BYTE

.code
LibMain proc hInstance:DWORD, reason:DWORD, reserved1:DWORD
    mov eax, 1
    ret
LibMain endp

Java_Simulator_1opt1_1or_1true1_Evalx proc JNIEnv:DWORD, jobject:DWORD, in_a:BYTE
    ; (a, true)or

    mov EAX, 0
    mov EBX, 0
    mov AL,in_a
    mov EBX,1
    or EAX, EBX
    ret
Java_Simulator_1opt1_1or_1true1_Evalx endp
End LibMain
```

Figure 12.3 the linker for MASM32 uses a definitions file to determine which symbols should be exported. Note that the library name does not require the replacing underscores. Additional exports are added on separate lines.

```
LIBRARY Simulator_opt1_or_true1
EXPORTS Java_Simulator_1opt1_1or_1true1_Evalx
```

Figure 12.3: Example definitions file for a dynamic link library (DLL)

Edit SimulatorGeneratorMasm32.java and run TestSimulatorGenerator-Masm32.java

This lab requires generating the instructions for the F-statement. Most of the other code is generated by the skeleton. To generate the instructions, the AST should be walked in post order as a simple topological sort and stored in the operations list. It is then necessary to assign signals and intermediate signals to registers and write instructions for each AST object. For example VarExpr or ConstantExpr can be converted into 8-bit immediate-mode mov instructions. The other operators can be implemented using various Boolean operator instruction. It is possible to implement all expressions including a NotExpr using a single instruction and no conditionals. The result of the last computation for the statement should be stored in EAX as the return value for the function. x86 has four 32-bit general purpose registers EAX, EBX, ECX, and EDX and supports 16-bit and 8-bit access modes. A helper class is provided which contains the names of that correspond to the other addressing modes (*i.e.*, AL references the low byte of EAX). Since there are only four registers (or fewer if you remove some from the Register.General list for debugging purposes), it is possible that large assignment statements will require the use of memory. A simple FIFO (registerQueue) strategy is used to determine which register to save to memory and reassign before performing an operation. Memory can be allocated using the LOCAL keyword: `LOCAL m0 :BYTE`. These allocation statements should be immediately following the function signature before any instructions. In order to track which memory is in use the hash map memoryMap maps the name of the memory allocation to a Boolean value of whether it is used. For convenience the IdentityHashMap storageMap maps an Expr to the name of either the memory or register currently storing its output value.

12.4 *Evaluation*

There is no automated evaluation for this lab due to the variety of possible system dependencies. You will have to meet with the course staff to be evaluated, which will include you answering questions about how your design works. Your generated assembly code should pass all of the tests from Lab 8.

There are potentially bonus marks if you find a cross-platform assembler with minimal system dependencies and develop a bit of a skeleton and instructional support for it. The MARS MIPS simulator mentioned above is the best possibility for this that we are aware of.

Appendix A
Extra VHDL *Lab Exercises*

A.1 Define-Before-Use Checker

Compilers generally perform some form of semantic analysis on the AST of the program after the input program is parsed. The analysis might include checking that all variables/signals are defined before they are used. In this part of the lab, write a define-before-use checked that traverses through a VHDL AST and determines whether all signals that are used in the signal assignment statements are defined before they are used. In addition, for all signal assignment statements, a signal that is being assigned (left-hand side) to an expression (right-hand side) must not be an input signal. Driving an input signal simultaneously from two sources would cause undefined behaviour at run time.

Within a design unit, a signal may be defined in the entity declaration as an input bit or output bit; it may also be defined in the optional signal declaration list within the body of the corresponding architecture. For example, if we consider the VHDL program shown in Figure A.1, the following signals are defined in this entity-architecture pair: `a0`, `b0`, `a1`, `b1`, `a2`, `b2`, `a3`, `b3`, `Cin`, `sum0`, `sum1`, `sum2`, `sum3`, `Cout`, `V`, `c0`, `c1`, `c2`, `c3`, and `c4`.

The define before use checker should throw an exception if it checks the program shown in Figure A.1. This figure illustrates the two violations your checker should detect:

a. The assignment statement in line 16 uses a signal called '`c`', which is undefind in this program.
b. The assignment statement in line 17 tries to assign an input pin/signal ('`Cin`') to an expression.

The checker should throw a `RuntimeException` exception upon the first violation that is encountered.

All of the code that you will write for the checker should be in the class `DefBeforeUseChecker` in the package `ece351.vhdl`.

`TestDefBeforeUseCheckerAccept` and `TestDefBeforeUseCheckerReject` are JUnit tests available for testing the checker. These two classes are found in the package `ece351.vhdl.test`.

A.1.1 Some Tips

- For VHDL programs that have multiple design units, apply the violation checks per design unit (i.e., treat them separately).
- Maintain the set of declared signals in such a way that it is easy to identify where the signals are declared in the design unit.

Figure A.1: VHDL program used to illustrate signal declarations and the use of undefined signals in signal assignment statements.

```
entity four_bit_ripple_carry_adder is port (
    a0, b0, a1, b1, a2, b2, a3, b3, Cin : in bit;
    sum0, sum1, sum2, sum3,Cout, V: out bit
);
end four_bit_ripple_carry_adder;


architecture fouradder_structure of four_bit_ripple_carry_adder is
     signal c1, c2, c3, c4: bit;
begin
    FA0 : entity work.full_adder port map(a0,b0,Cin,sum0,c1);
    FA1 : entity work.full_adder port map(a1,b1,c1,sum1,c2);
    FA2 : entity work.full_adder port map(a2,b2,c2,sum2,c3);
    FA3 : entity work.full_adder port map(a3,b3,c3,sum3,c4);

    V <= c xor c4;
    Cin <= c4;
end fouradder_structure;
```

A.2 Inline $\mathcal{F}$ intermediate variables

```
X <= A or B;
Y <= X or C;
...
Y <= (A or B) or C;
```

Appendix B
Advanced Programming Concepts

These are some advanced programming ideas that we explored in the labs.

B.1 Immutability

Pragmatics: AST classes

Immutability is not a 'design pattern' in the sense that it is not in the design patterns book.[1] It is, however, perhaps the most important general design guideline in this section. Immutability has a number of important advantages:

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass., 1995

- Immutable objects can be shared and reused freely. There is no danger of race conditions or confusing aliasing. Those kinds of bugs are subtle and difficult to debug and fix.
- No need for defensive copies. Some object-oriented programming guidelines advocate for making defensive copies of objects in order to avoid aliasing and race condition bugs.
- Representation invariants need to be checked only at object construction time, since there is no other time the object is mutated.

There are also a number of possible disadvantages:

- Object trees must be constructed bottom-up. This discipline can be useful for understanding the program, but sometimes requires a bit of adjustment for programmers who are not familiar with this discipline.
- Changing small parts of large complex object graphs can be difficult. If the object graphs are trees then the visitor design pattern can be used to make the rewriting code fairly simple.
- Sometimes data in the problem domain are mutable, and so are best modelled by mutable objects. A bank account balance is a classic example.

B.2 Representation Invariants

Tiger: repOk() methods

Representation invariants are general rules that should be true of all objects of a particular class. For example:

- LinkedList.head != null
- LinkedList.size is the number of nodes in the list
- that the values in a certain field are sorted

These rules are often written in a repOk() method, which can be called at the end of every method that mutates the object (for immutable objects these methods are just the constructors). It is a good idea to check these rules right after mutations are made so that the failure (bad behaviour) of the software occurs close to the fault (error in the source code). If these rules are not checked right away, then the failure might not occur until sometime later when the execution is in some other part of the program: in those circumstances it can be difficult to find and fix the fault.

B.3 Functional Programming

Some ideas from functional programming that are useful in the object-oriented programming context:

- immutability
- recursion
- lazy computation
- higher-order functions
- computation as a transformation from inputs to outputs

In w2013 we've talked about immutability, and the code has embodied both immutability and computation as a transformation from inputs to outputs.

See the *Design Patterns* book or Wikipedia or SourceMaking.com

Appendix C *Design Patterns*

The material in this section is mostly copied from SourceMaking.com.

Design patterns represent a standard set of ideas and terms used by object-oriented programmers. If you get a job writing object-oriented code these are concepts and terms you will be expected to know.

Design patterns[1] are classified into three groups $\left\{\begin{array}{l}\textit{creational}\\ \textit{structural}\\ \textit{behavioural}\end{array}\right.$

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass., 1995

C.1 Iterator

- Behavioural
- Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
- Enables decoupling of collection classes and algorithms.
- Polymorphic traversal

C.2 Singleton

Pragmatics: ConstantExpr

- Creational
- Ensure a class has only predetermined small number of named instances, and provide a global point of access to them.
- Encapsulated 'lazy' ('just-in-time') initialization.

C.3 Composite

Pragmatics: Expr class hierarchy

- Structural
- Compose objects into tree structures to represent whole-part hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
- Recursive composition
- *e.g.*, 'Directories contain entries, each of which could be a file or a directory.'

C.4 Template Method

Pragmatics: BinaryExpr.simplifySelf()

- Behavioural
- Define the skeleton of an algorithm in an operation, deferring some steps to client subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
- Base class declares algorithm 'placeholders', and derived classes implement the placeholders.

C.5 Interpreter

Pragmatics: Expr.simplify()

- Behavioural
- Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
- Map a domain to a language, the language to a grammar, and the grammar to a hierarchical object-oriented design.

Tiger: §4.3 + lab manual

- In other words, add a method to the AST for the desired operation. Then implement that method for every leaf AST class.
- An alternative to Visitor. Interpreter makes it easier to modify the grammar of the language than Visitor does, but with Visitor it is easier to add new operations.

C.6 Visitor

Pragmatics: Expr class hierarchy

- Behavioural
- Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.
- Do the right thing based on the type of two objects.
- Double dispatch
- An alternative to Interpreter.

Appendix D
Bibliography

[1] A. W. Appel and J. Palsberg. *Modern Compiler Implementation in Java*. Cambridge University Press, 2004.

[2] J. Bloch. *Effective Java*. Addison-Wesley, Reading, Mass., 2001.

[3] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, Aug. 1986.

[4] B. Eckel. *Thinking in Java*. Prentice-Hall, 2002. http://www.mindview.net/Books/TIJ/.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass., 1995.

[6] B. Liskov and J. Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley, Reading, Mass., 2001.

[7] J. C. Reynolds. User-defined types and procedural data as complementary approaches to data abstraction. In S. A. Schuman, editor, *New Directions in Algorithmic Languages: IFIP Working Group 2.1 on Algol*. INRIA, 1975.

[8] M. L. Scott. *Programming Language Pragmatics*. Morgan Kaufmann, 3 edition, 2009.

[9] M. Torgersen. The Expression Problem Revisited: Four new solutions using generics. In M. Odersky, editor, *Proc.18th ECOOP*, volume 3344 of *LNCS*, Oslo, Norway, June 2004. Springer-Verlag.

[10] P. Wadler. The expression problem, Nov. 1998. Email to Java Generics list.

[11] W. Wulf and M. Shaw. Global variables considered harmful. *ACM SIGPLAN Notices*, 8(2):80–86, Feb. 1973.

[12] M. Zenger and M. Odersky. Independently extensible solutions to the expression problem. In *Proc.12th Workshop on Foundations of Object-Oriented Languages*, 2005.