DEREK RAYSIDE & WALLACE WU

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.
- TA's Jon Eyolfson, Vajih Montaghami, and Alireza Mortezaei.

Contents

See the Appendices for a brief discussion of background material, computing environment, tools, *etc.*

Concepts: regular languages, regular expressions, EBNF, recognition *vs.* parsing, recursive descent, tokenizing/lexing, pretty-printing, abstract syntax tree (AST), representation invariants, object comparisons, test harnesses

Concepts: trees, transformations, XML, SVG, immutability, equality, isomorphism, equivalence, type tests, casting,DOM *vs.* SAX parser styles

Concepts: predict sets, intermediate forms, optimization, inheritance, polymorphism, variables *vs.* objects, type tests, casting, memory safety, composite design pattern, template design pattern, singleton design pattern, recursive functions, recursive structures, higher-order functions, associativity, commutativity, precedence, equivalence of logical formulae

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

List of Figures

Reflections

See the Appendices for a brief discussion of background material, computing environment, tools, *etc.*

Overview

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

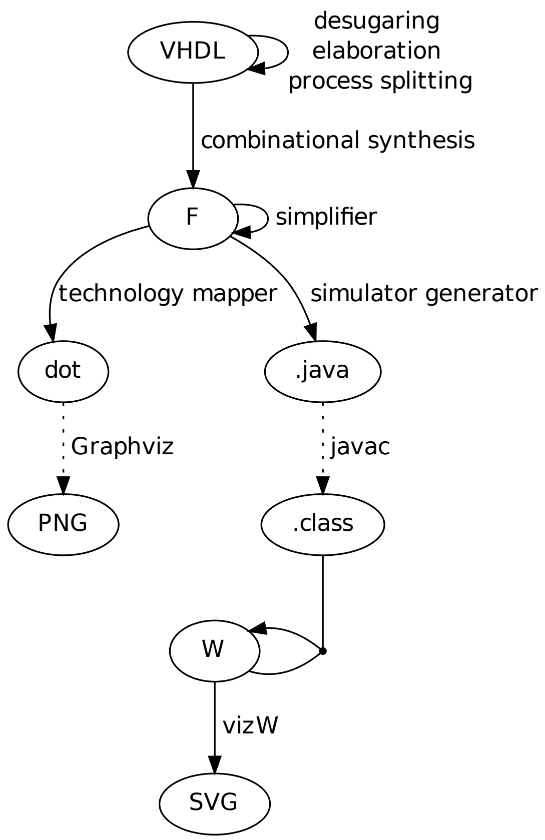

Figure 1: Overview of ECE351 Labs.
Nodes represent file types (*e.g.*, VHDL). All of these file types, with the exception of .class and PNG files, are text files.
Edges represent translators between different file types. Solid edges represent translators that we will implement in ECE351. Dotted edges 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.

0.1 Pre-Lab: Computing Environment

The door code for E2-2363 is 5 1 2 3+4

0.1.1 Our Environment

Follow the instructions at ecegit.uwaterloo.ca/getting-started/ 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.

0.1.2 Getting Your Code

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

- ☐ `mkdir ~/git`
- ☐ `cd ~/git`
- ☐ `git clone git@ecegit.uwaterloo.ca:USERNAME/ece351`
- ☐ `cd ece351`
- ☐ `git remote add skeleton git@ecegit.uwaterloo.ca:ece351`
- ☐ `git fetch skeleton`
- ☐ `git checkout -b skeleton skeleton/master`
- ☐ `git checkout master`

replace `USERNAME` with your username

Now we need to initialize the tests submodule:

- ☐ `git submodule init`
- ☐ `git submodule update`
- ☐ `cd tests`
- ☐ `git checkout master`

0.1.3 Checking for Updates

Within your repository you can check for updates using the following commands:

- ☐ `git checkout skeleton`
- ☐ `git pull`
- ☐ `git checkout master`

If there are any updates you'll see files get modified. Doing `git pull` automatically updates from the remote `skeleton` and merges any changes in your `skeleton` branch. **You should always checkout back to master since you don't want to modify the skeleton branch, only use it to get changes from us**.

You'll also want to check for updates to the test input files:

- ☐ `cd tests`
- ☐ `git checkout master`
- ☐ `git pull`
- ☐ `cd ..`
- ☐ `git add tests`
- ☐ `get commit -m 'updating tests submodule pointer'`

0.1.4 *Updating Your Repository*

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

- ☐ `git merge skeleton`

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.

0.1.5 *Commiting Changes*

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

0.1.6 *Uploading Your Repository*

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

☐ `git push`

0.1.7 *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] There is also a quiz to take on Learn.

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

Here is a checklist:

☐ Took quiz on Learn

☐ Got updates from the skeleton

☐ Merged updates into your master branch

☐ Edit `meta/hours.txt`

☐ Uploaded your changes

0.1.8 *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 4.2.1 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` (or wherever your repository is

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

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

☐ Click **Finish**

0.1.9 *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

0.2 *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.

0.3 *Object-Oriented Programming*

You will need to understand object-oriented programming, 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,[2] 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] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass., 1995

0.4 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.

Concepts: regular languages, regular expressions, EBNF, recognition *vs.* parsing, recursive descent, tokenizing/lexing, pretty-printing, abstract syntax tree (AST), representation invariants, object comparisons, test harnesses

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

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 2.1 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	$\rightarrow$	(`[A-Za-z]`)$^+$
Bits	$\rightarrow$	('0' | '1')$^+$

Figure 1.2: Grammar for $\mathcal{W}$. Understand with standard lexical conventions: whitespace is not required around punctuation (*e.g.*, ':', ';'), but is required between zeroes and ones. By convention we will call the top production *Program*.

1.1 Before You Get Started

You may notice your testing directory is empty. Your testing directory is actually another git repository that is linked in to your repository as a *submodule*. This linking makes it easier for the entire class to share test inputs. But right now your tests directory is empty because the submodule hasn't been initialized, so let's do that by executing the following commands from your ece351 folder:

Git submodules are powerful and useful but also a little bit complicated. If you just pull the changes from the staff everything should be simple for you. You do have the option of making your own branch of the tests repository and creating new test cases there to share with the staff and your classmates. Speak with the course staff for help with this if necessary.

```
$ git submodule init
$ git submodule update
$ cd tests
$ git checkout master
```

Before you start working on a lab you should make sure that you pull the latest changes from the skeleton code by following the steps in the pre-lab exercise. In addition to those steps you'll want to make sure you have the most recent tests by executing git pull inside the tests directory. Finally, make sure that you're reading the latest version of this lab manual.

1.2 *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*.

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.

Another potential issue for Windows users is the difference between line breaks on Windows and *nix. We'll have this sorted out shortly.

1.3 *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 production in the grammar. For recognizers these methods return void.

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

Execution of these production 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 production 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.

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

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

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.

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

1.4.1 *Introducing ImmutableList*

We choose to make our AST objects immutable because mutation opens the door to many kinds of difficult to fix bugs and because we do not need the benefits of mutation for our AST objects.

We use the ImmutableList[1] in our AST objects when we need a list of sub-objects. The standard ArrayList and LinkedList classes in the JDK library are mutable and so we avoid them here.

[1] From the Parboiled parser generator library. We'll use Parboiled to write parsers in subsequent labs.

Figure 1.3: Simple example of using Parboiled's ImmutableList class

```
// get a reference to the empty list
final ImmutableList emptyList = ImmutableList.of();
// make a new list with one element in it
final ImmutableList singletonList = empytList.append("data");
// iterate over and print the elements in a list
for (final Object obj : singletonList) {
    System.out.println(obj);
}
```

1.5 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, reparse 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.TestWRDParserAccept
ece351.w.rdescent.TestWRDParserReject

1.6 Reading

1.6.1 Tiger Book[2]

[2] 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

1.6.2 Programming Language Pragmatics[3]

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

- 2.2 Scanning
- 2.3.1 Recursive Descent

1.6.3 Web Resources

Thinking in Java [4]

[4] B. Eckel. *Thinking in Java*. Prentice-Hall, 2002

Prof Alex Aiken @ Stanford on recursive descent:
http://www.youtube.com/watch?v=Kdx5HOtb-Zo
http://www.youtube.com/watch?v=S7DZdn33eFY

Concepts: trees, transformations, XML, SVG, immutability, equality, isomorphism, equivalence, type tests, casting,DOM *vs.* SAX parser styles

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.

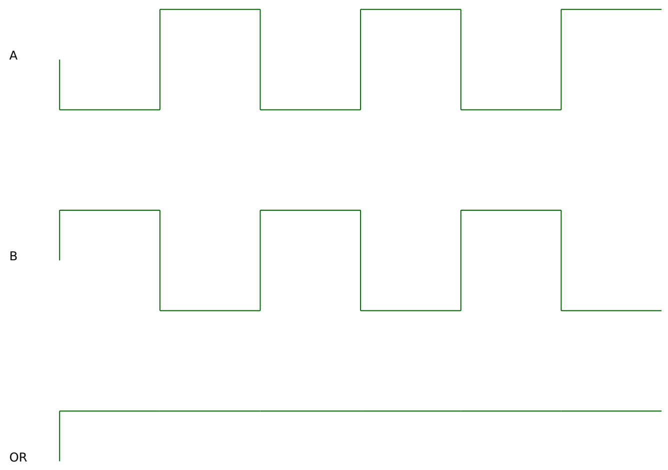

Figure 2.2: Rendered SVG of $\mathcal{W}$ file from Figure 2.1

```
<?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.TestW2SVGtransform

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

2.2 *Write object comparisons for the SVG waveforms*

In the previous step you wrote a translator from $\mathcal{W}$ to SVG and manually inspected the output to see that it was sensible. Is it possible to evaluate the output mechanically rather than manually? Yes. That's what we're going to do next. We will consider three different levels of comparison:

Sources:
ece351.w.svg.WSVG
ece351.w.svg.Line
ece351.w.svg.Pin
Libraries:
ece351.util.Examinable
ece351.util.Examiner
ece351.util.ExaminableProperties
Tests:
ece351.w.svg.TestW2SVGisomorphic
ece351.w.svg.TestW2SVGequivalent

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).

isomorphic: We will say that two objects are *isomorphic* if they have the same elements, perhaps arranged in a different order. What this means exactly for each class of object needs to be defined. In this lab we will consider that two SVG files representing $\mathcal{W}$ programs

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

are isomorphic if they have all the same line segments drawn in the same locations on the screen. The order in which these line segments appear in the SVG file may differ. Any two objects that are equals are also isomorphic.

equivalent: We will say that two objects are *equivalent* if they mean the same thing but possibly have some superficial differences in appearance. For example, we might consider that the phrases *won't* and *will not* have the same meaning even though they look different. What equivalence means for rich object structures needs to be precisely defined in each case. In this lab we will consider that two SVG files representing $\mathcal{W}$ programs are equivalent if the line segments of one can be produced by a geometric translation[2] of the line segments of the other. Any two objects that are isomorphic are also equivalent.

[2] *i.e.*., sliding them around on the canvas

Each of these three comparisons define their own mathematical *equivalence class*. A mathematical equivalence class has three properties:

reflexive: x.equals(x)

symmetric: x.equals(y) $\Leftrightarrow$ y.equals(x)

transitive: x.equals(y) && y.equals(z) $\Rightarrow$ x.equals(z)

Additionally, there is a hierarchical relationship between our equivalence classes: equality implies isomorphism, and isomorphism implies equivalence. The ExaminableProperties class contains utility methods to check these general properties.

Your task is to implement these three comparison operations for the WSVG, Line, and Pin classes. These comparison operations are how we will assess whether your $\mathcal{W}$ to SVG transformation works properly.

2.3 Reading

2.3.1 Object Contract: Equality and Hashing

http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#equals(java.lang.Object)
Effective Java [3]
http://www.artima.com/lejava/articles/equality.html
http://www.angelikalanger.com/Articles/JavaSolutions/SecretsOfEquals/Equals.html
Implementation of java.util.AbstractList.equals()
Implementation of ece351.util.Examiner

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

Not yet discussed: default implementations, inheritance, cyclic object structures

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

Concepts: predict sets, intermediate forms, optimization, inheritance, polymorphism, variables *vs.* objects, type tests, casting, memory safety, composite design pattern, template design pattern, singleton design pattern, recursive functions, recursive structures, higher-order functions, associativity, commutativity, precedence, equivalence of logical formulae

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	$\rightarrow$	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 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 *call 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.

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:

```
program()
    formula()
        var()
            id()
        '<='
        expr()
            term()
                factor()
                    var()
                        id()
            'or'
            term()
                factor()
                    var()
                        id()
        ';'
```

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.

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.

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.

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

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

Sources:
ece351.f.rdescent.FRecursiveDescentRecognizer
Libraries:
ece351.util.Lexer
ece351.util.CommandLine
Tests:
ece351.f.rdescent.TestFRDRecognizer

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.

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

3.4 Write object comparisons for the $\mathcal{F}$ AST classes

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.*

Sources:
ece351.common.ast.VarExpr
ece351.common.ast.ConstantExpr
ece351.f.ast.FProgram
ece351.common.ast.AssignmentStatement
ece351.common.ast.UnaryExpr
ece351.common.ast.CommutativeBinaryExpr
Libraries:
ece351.util.Examinable
ece351.util.Examiner
ece351.util.ExaminableProperties
Tests:
use the parser tests below

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

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

- 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.

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

Reflect: What classes in the AST code do not have a corresponding production in the grammar? Why?

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.

Reflect: Why are some fields in the AST classes not private?

3.6 Background: Object-Oriented Programming

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

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.

3.7 Reading

See also the readings for LAB1 above.

3.7.1 Tiger Book[1]

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

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

3.7.2 Design Patterns[2] & Programming Guidelines

Composite Design Pattern
http://en.wikipedia.org/wiki/Composite_pattern
http://sourcemaking.com/design_patterns/composite
http://userpages.umbc.edu/~tarr/dp/lectures/Composite.pdf

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

Template Design Pattern
http://en.wikipedia.org/wiki/Template_method_pattern
http://sourcemaking.com/design_patterns/template_method

Abstract Superclass Rule [3,4] Classes should be either *abstract* (*i.e.,* designed for inheritance), or *final* (*i.e.,* preventing subclasses)

[3] W. L. Hürsch. Should superclasses be abstract? In M. Tokoro and R. Pareschi, editors, *Proc.9th ECOOP*, volume 821 of *LNCS*, pages 12 – 31, Bologna, Italy, July 1994. Springer-Verlag

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

Thinking in Java [5] is a good resource for object-oriented programming in general and Java in particular. It does a good job of explaining inheritance and polymorphism.

[5] B. Eckel. *Thinking in Java*. Prentice-Hall, 2002

1 Introduction to Objects
6 Reusing Classes
7 Polymorphism

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

Concepts: intermediate languages, associativity, commutativity, precedence, identity element, absorbing element, equivalence of logical formulae, term rewriting, template design pattern

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

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

Compilers usually perform their optimizations on programs in 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 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.

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 4.2 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 *We are not implementing equivalent() for boolean formulas*

$\mathcal{F}$ is a language of boolean formulas. Checking equivalence of boolean formulas is an NP-complete problem. Therefore, we will not attempt to do this because of the computational complexity. Instead we will define the equivalent() method to simply call the isomorphic() method.

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

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: $2^3 = 8$.

We will pursue an approach that is computable in polynomial time, and hence much faster than truth table construction. However, there are some pairs of formulas that our approach will not be able to detect as equivalent.

There has been a lot of clever research[1] on how to do equivalence testing of boolean formulas. We are going to explore a fairly simple approach based on term-rewriting. There are other approaches that you may challenge yourself with for participation marks.

[1] *e.g.*, binary decision diagrams

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

4.2 *Write standardize methods for the* AST *classes*

Sources:
ece351.common.ast.NaryAndExpr
ece351.common.ast.NaryOrExpr
ece351.common.ast.NaryExpr
Tests:
ece351.f.TestSimplifier

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.

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. Subtraction, for example, is left associative: $x - y - z$ is equivalent to $(x - y) - z$ and not $x - (y - z)$.

See the course notes or Wikipedia or the text book for a longer discussion of associativity.

conjunction = and

disjunction = or

Detecting the equivalence of $x+(y+z)$ and $(x+y)+z$ is tricky because these parse trees are not structurally similar (recall that 'structually 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: a tree transformation.

Figure 4.1 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 equivalence.

This transformation is performed by the *NaryExpr.standardize()* method, which is where most of the code for this step needs to be written.

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.

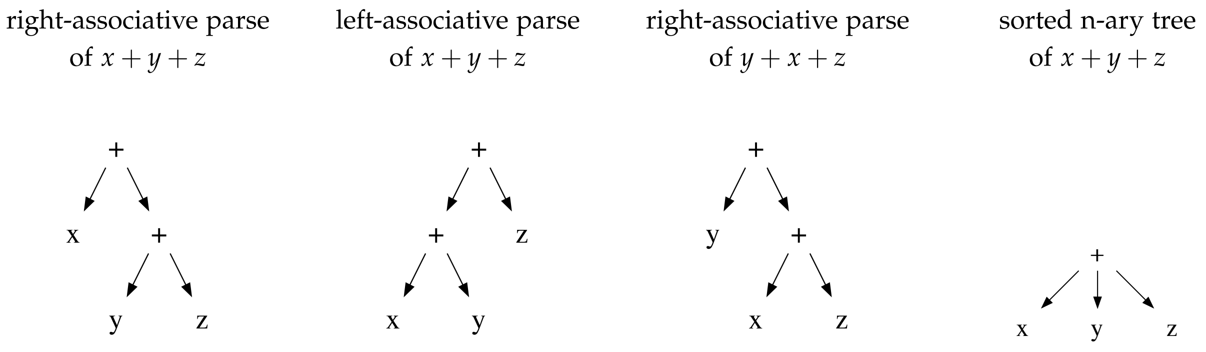
right-associative parse of $x+y+z$
left-associative parse of $x+y+z$
right-associative parse of $y+x+z$
sorted n-ary tree of $x+y+z$

Figure 4.1: Four trees that represent logically equivalent circuits. The sorted n-ary representation makes equivalence comparisons easier.

4.3 *Write simplify methods for the* AST *classes*

Sources:
ece351.util.CommandLine.FSimplifierOptions
ece351.common.ast.NotExpr
ece351.common.ast.NaryExpr
Tests:
ece351.f.TestSimplifier
Libraries:
org.parboiled.common.ImmutableList
java.util.List
java.util.ArrayList
Concepts:
identity element
absorbing element
fixed point
iteration to a fixed point

Figure 4.2 lists the simplifications that we are going to apply to $\mathcal{F}$ programs, stratified into levels. Start implementing from the lowest level and work your way up. The last level, *absorption*, is a bonus and is not required.

Level 0 is standardization — the conversion of conjunction and disjunction nodes from from binary to n-ary discussed above. After standardization the expression AST should comprise only NaryAndExpr, NaryOrExpr, NotExpr, ConstantExpr, and VarExpr nodes. Therefore, all the remaining simplifications are implemented in these classes (or some common ancestor they share, such as NaryExpr).

The test harness runs the test inputs in the order of the levels. So once you've completed the standardization operation then your code should pass the first few tests, and so on. If you do not complete the bonus level (absorption) then the tests for that level will fail, but don't worry about that since they are not required to pass.

Level	**Description**	**Code**
0	Standardize	standardize() method from above
1	Constant Folding	NotExpr, NaryExpr $!0 = 1$ $!1 = 0$ $x \cdot 0 = 0$ $x \cdot 1 = x$ $x + 0 = x$ $x + 1 = 1$
2	Complement	NotExpr, NaryExpr $!!x = x$ $x \cdot !x = 0$ $x + !x = 1$
3	Deduplication	NaryExpr $x + x = x$ $x \cdot x = x$
4	Absorption	NaryAndExpr, NaryOrExpr $x + (x \cdot y) = x$ $x \cdot (x + y) = x$

Figure 4.2: Optimizations for $\mathcal{F}$ programs, stratified into levels and described by algebraic identities where possible.

Absorption is a bonus exercise — not required for this lab. There are two possible interpretations of the identity given in the table: First, that x is a VarExpr. Second, that x is an NaryExpr. The latter case is harder.

4.4 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.[2] The type tests are implicit in the dynamic dispatch performed when the primitive operations are called.

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

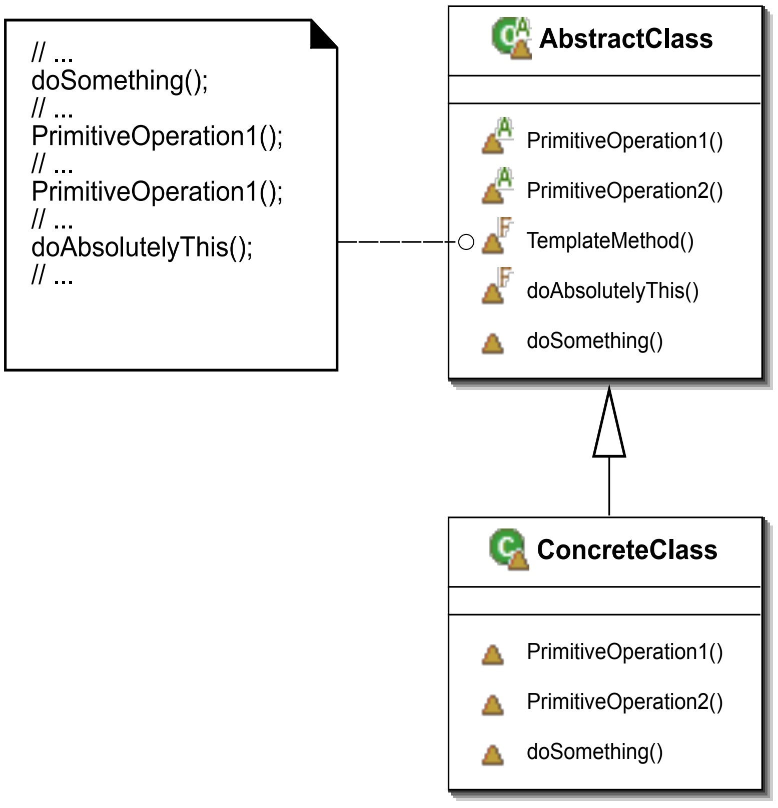

Figure 4.3: Template Design Pattern illustrated by UML Class Diagram. Image from http://en.wikipedia.org/wiki/File:Template_Method_UML.svg. Licensed under GNU Free Documentation Licence.

Expr:

Primitive Op	*Implemented By*	*Template Method*
operator()	every subclass of Expr	toString()

NaryExpr:

Primitive Op	*Implemented By*	*Template Method*
getAbsorbingElement()	NaryAndExpr, NaryOrExpr	simplifyOnce()
getIdentityElement()	NaryAndExpr, NaryOrExpr	simplifyOnce()
getCorrespondingBinaryExprClass()	NaryAndExpr, NaryOrExpr	standardize()
newNaryExpr()	NaryAndExpr, NaryOrExpr	standardize(), simplifyOnce()
simplifyOnce()	NaryAndExpr, NaryOrExpr	simplify()
simplifySelf()	NaryAndExpr, NaryOrExpr	simplifyOnce()

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.

Concepts: domain specific languages (DSLs): internal *vs.* external, Parsing Expression Grammars (PEGs), whitespace, debugging with parboiled, 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 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. We are implementing less functionality in this lab than we did in LAB1. In this lab we will just write a recognizer and parser for $\mathcal{W}$: we will reuse your $\mathcal{W} \rightarrow$SVG transformer from LAB1. 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.

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()	
	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 $\mathcal{F}$. 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 grammar for $\mathcal{F}$ above in 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.

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. 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 productions 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 snipped as in Figure 5.1.

```
public Rule List() { return ZeroOrMore(Sequence(Name(), W1())); }
public Rule Name() { return OneOrMore(Letter()); }
public Rule Letter() { return CharRange('A', 'Z'); }
```

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

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 5.4 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

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.

a breakpoint inside that method. Setting a breakpoint inside a rule constructor will not do what you want.

```
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
        );
    }
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]
        swap(),                                         // swap top two elements on stack: [String, List]
        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 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 5.4.

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 '.

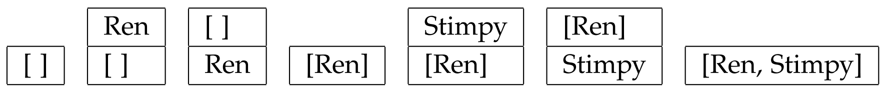

Figure 5.3: Stack of Parboiled parser from Figure 5.2 while processing input 'Ren Stimpy '. Time proceeds left to right. Think about which state of the stack corresponds to which line of the parser source code.

5.3.1 *The stack for our $\mathcal{W}$ parser will have two lists*

The stack for our $\mathcal{W}$ programs will have two lists active at any given time: the list of waveforms / lines of input / pins, and the list of bits for the current waveform / pin / line of input. The bit list will always be closer to the top of the stack and the waveform list will always be closer to the bottom of the stack.

```
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 CharRange('A', 'Z'); }  // rule from recognizer
```

Figure 5.4: 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.

5.4 *Reading*

Parboiled:
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

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).

Concepts: Parsing Expression Grammars (PEGs), whitespace

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

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

Files:
ece351.f.parboiled.FParboiledRecognizer.java
ece351.f.parboiled.FParboiledParser.java
Tests:
ece351.f.parboiled.TestFParboiledRecognizer
ece351.f.parboiled.TestFParboiledParser

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

Program	→	Formula*
Fomula	→	Var '<=' Expr ';'
Expr	→	Term ('+' Term)*
Term	→	Factor ('.' Factor)*
Factor	→	'!' Factor | '(' Expr ')' | Var | Constant
Constant	→	'0' | '1'
Var	→	id

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

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 in in the previous lab 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.

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, if desired.

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

Concepts: visitor design pattern, interpreter design pattern, tree traversals, common subexpression elimination, hash structures, hashing and (non-)determinism of iteration order, IdentityHashMap

Files:
ece351.f.TechnologyMapper
ece351.f.ast.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 + 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="plain"];
    // circuit: X = A + B
    or1 [label="", image="or_noleads.png"];
    A -> or1;
    B -> or1;
    or1 -> X;
}
```

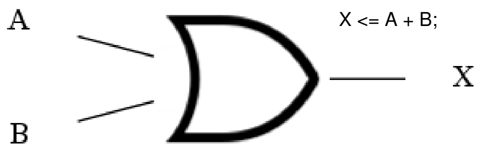

Figure 7.1: Example Graphviz input file and rendered output for $\mathcal{F}$ program `X <= A + B;`

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 Introducing the visitor design pattern

The *visitor* design pattern[1] is commonly used when writing translators. Parse trees and abstract syntax trees are complex composite[2] object structures, and we want to perform a variety of operations on these structures. For example, the previous labs we applied algebraic laws to ASTs representing $\mathcal{F}$ expressions.

[1] 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.

[2] We discussed the Composite design pattern previously.

There are two main strategies for adding new operations to an AST: either adding new methods to the AST classes themselves, or writing the new operation in its own separate new class. The former is known as the *interpreter* design pattern and is what we did for the simplify, collect, and distribute operations in the previous labs. The latter is known as the *visitor* design pattern and is what we will use in this lab.

The visitor design patern 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 7.2.

```
public abstract class FExprVisitor extends ExprVisitor {

        // Any FExprVisitor subclass must implement the following methods
        public abstract Expr visit(ConstantExpr e);
        public abstract Expr visit(VarExpr e);
        public abstract Expr visit(NotExpr e);
        public abstract Expr visit(AndExpr e);
        public abstract Expr visit(OrExpr e);
        public abstract Expr visit(NaryAndExpr e);
        public abstract Expr visit(NaryOrExpr e);
```

Figure 7.2: Abstract super class for visitors for $\mathcal{F}$ expression ASTs.
There is a visit method for each concrete $\mathcal{F}$ AST class.

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. You can read more about this idea online.

To the FExpr class we add the signature for the accept method, which is then implemented in each concrete AST class as a call to a visit method, as shown in Figure 7.3.

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

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

Figure 7.3: Signature and implementation of accept method. The signature belongs in the Expr class and the implementation is put in each concrete subclass of Expr.

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 7.4 and 7.5).

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

```
/**
 * Dispatch to traverse(NaryExpr) and traverse(BinaryExpr)
 * and traverse(UnaryExpr).
 * */
public Expr traverse(final Expr e) {
        if (e instanceof NaryExpr) {
                traverse( (NaryExpr) e );
        } else if (e instanceof BinaryExpr) {
                traverse( (BinaryExpr) e );
        } else if (e instanceof UnaryExpr) {
                traverse( (UnaryExpr) e );
        } else {
                e.accept(this);
        }
        return e;
}
public abstract Expr traverse(final NaryExpr e);
public abstract Expr traverse(final BinaryExpr e);
public abstract Expr traverse(final UnaryExpr e);
```

Figure 7.4: The traverse methods of FExprVisitor

We can write visitors that perform a number of useful tasks for this lab. For example, Figure 7.6 lists a visitor that builds a set of all the nodes in an $\mathcal{F}$ expression AST.

Figure 7.5: Implementation of PostOrderFExprVisitor. Any visitors that extend this class will visit the nodes of an $\mathcal{F}$ expression AST in pre-order (*i.e.*, parents before children). It would be easy to make a variant of this class that visited the AST nodes post-order (parents after children).

```
package ece351.f.ast;

import ece351.common.ast.*;

public abstract class PostOrderFExprVisitor extends FExprVisitor {

    @Override
    public Expr traverse(final NaryExpr e) {
        // <snip>
        for (final Expr c : e.children) {
            traverse(c);
        }
        e.accept(this);
        // </snip>
        return e;
    }

    @Override
    public Expr traverse(final BinaryExpr b) {
        // <snip>
        traverse(b.left);
        traverse(b.right);
        b.accept(this);
        // </snip>
        return b;
    }

    @Override
    public Expr traverse(final UnaryExpr u) {
        // <snip>
        traverse(u.expr);
        u.accept(this);
        // </snip>
        return u;
    }
}
```

Figure 7.6: 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 PostOrderFExprVisitor {

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

        private ExtractAllExprs(final Expr e) { traverse(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 visit(ConstantExpr e) { exprs.add(e); return e; }
        @Override public Expr visit(VarExpr e) { exprs.add(e); return e; }
        @Override public Expr visit(NotExpr e) { exprs.add(e); return e; }
        @Override public Expr visit(AndExpr e) { exprs.add(e); return e; }
        @Override public Expr visit(OrExpr e) { exprs.add(e); return e; }
        @Override public Expr visit(XOrExpr e) { exprs.add(e); return e; }
        @Override public Expr visit(NAndExpr e) { exprs.add(e); return e; }
        @Override public Expr visit(NOrExpr e) { exprs.add(e); return e; }
        @Override public Expr visit(XNOrExpr e) { exprs.add(e); return e; }
        @Override public Expr visit(EqualExpr e) { exprs.add(e); return e; }
        @Override public Expr visit(NaryAndExpr e) { exprs.add(e); return e; }
        @Override public Expr visit(NaryOrExpr e) { exprs.add(e); return e; }
}
```

7.2 *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.

ece351.f.TechnologyMapper

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.

7.3 *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.

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 it's 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.

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.

The type of TechnologyMapper.nodes should probably be changed to TreeSet so that the nodes are printed in alphabetical order.

7.4 *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 + B appears twice:

```
X <= D . (A + B);
Y <= E . (A + 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 7.7 shows the circuit produced by a synthesizer that does not perform common subexpression elimination: there are two OR gates that both compute A + B. A more sophisticated synthesizer that does perform common subexpression elimination would synthesize a circuit like the one in Figure 7.8 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 + B) . !(A + 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+B: $(A+B)_1$, $(A+B)_2$, and $(A+B)_3$. These three expressions are in the same *equivalence class* according to our equivalent() method. We select one representative from this group, say $(A+B)_2$, and use that to represent all of the equivalent expressions. So our substitution table will contain the tuples $\langle (A+B)_1, (A+B)_2 \rangle$, $\langle (A+B)_2, (A+B)_2 \rangle$, $\langle (A+B)_3, (A+B)_2 \rangle$. In other words, we'll use $(A+B)_2$ in place of $(A+B)_1$ and $(A+B)_3$.

   This substitution table can be built in the following way. First, compare every FExpr in the program with every other FExpr with the equivalent() 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.

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.

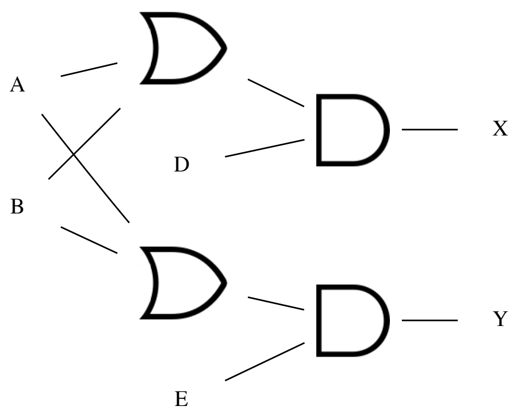

Figure 7.7: Synthesized circuit *without* common subexpression elimination for $\mathcal{F}$ program X <= D . (A + B); Y <= E . (A + B). There are two gates synthesized that both compute A + B.

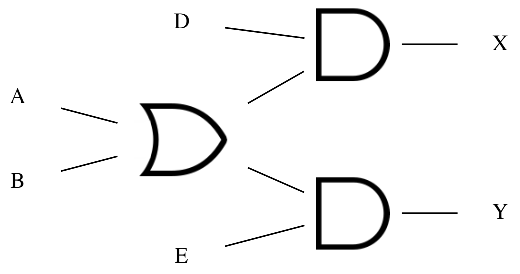

Figure 7.8: Synthesized circuit *with* common subexpression elimination for $\mathcal{F}$ program X <= D . (A + B); Y <= E . (A + B). There is only one gate that computes the value A + B, and the output of this gate is fed to two places.

7.4.1 *Evaluation*

The goal is to produce an equivalent circuit with the fewest possible gates. It is expected that about half the class will get as far as implementing common subexpression elimination as described above (which is all that the staff solution implements). If you wanted to work beyond that for bonus points there are a few possible ways you could do that, including but not limited to:

- Improving Expr.equivalent() to take DeMorgan's Laws or other algebraic identities into account.
- Looking for subsets of inputs that are common to more than two NaryExprs.
- 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). 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-hard, 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.

Concepts: program generation, preorder traversal

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

Files:
 ece351.f.SimulatorGenerator
 ece351.f.ast.PreOrderExprVisitor
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.

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)`.

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.

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;
        // write the input
        // 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 a = wprogram.valueAtTime("a", time);
            final boolean b = wprogram.valueAtTime("b", time);
            // values of output variables at this time step
            final String output_x = x(a, b) ? "1 " : "0 ";
            // store outputs
            output.get("x").append(output_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.

Figure 8.2: Implementation of DetermineInputVars

```
public final class DetermineInputVars extends PostOrderFExprVisitor {
        private final Set<String> inputVars = new LinkedHashSet<String>();
        private DetermineInputVars(final AssignmentStatement f) { traverse(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));
                }
                return Collections.unmodifiableSet(vars);
        }
        @Override public Expr visit(final ConstantExpr e) { return e; }
        @Override public Expr visit(final VarExpr e) { inputVars.add(e.identifier); return e; }
        @Override public Expr visit(final NotExpr e) { return e; }
        @Override public Expr visit(final AndExpr e) { return e; }
        @Override public Expr visit(final OrExpr e) { return e; }
        @Override public Expr visit(final XOrExpr e) { return e; }
        @Override public Expr visit(final NAndExpr e) { return e; }
        @Override public Expr visit(final NOrExpr e) { return e; }
        @Override public Expr visit(final XNOrExpr e) { return e; }
        @Override public Expr visit(final EqualExpr e) { return e; }
        @Override public Expr visit(NaryAndExpr e) { return e; }
```

Lab 9
VHDL *Recognizer & Parser*

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

Tests:
ece351.vhdl.test.TestVRecognizerAccept
ece351.vhdl.test.TestVRecognizerReject
ece351.vhdl.test.TestVParser
ece351.vhdl.test.TestDefBeforeUseCheckerAccept
ece351.vhdl.test.TestDefBeforeUseCheckerReject
ece351.vhdl.test.TestDeSugarer

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

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 'sig-

[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

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	→	*Relation* (*LogicOp Relation*)*
Relation	→	*Factor* (*RelOp* Factor)?
Factor	→	(**‘not’**)? *Literal*
Literal	→	*Id* | ‘’’ *Constant* ‘’’ | **‘(’** *Expr* **‘)’**
LogicOp	→	**‘and’** | **‘or’** | **‘xor’** | **‘nand’** | **‘nor’** | **‘xnor’**
RelOp	→	**‘=’**
Constant	→	**‘0’** | **‘1’**
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. Finally, Wallace Wu (TA) refactored Aman's code, which is the basis for the current lab.

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

nal' and 'SIGNAL' represent the same keyword in the language) and consume the appropriate amount of whitespace after the keywords are matched. For brevity, the keywords used in the VHDL grammar shown in Figure 9.1 are all written in lowercase characters; however, your recoginzer and parser should accept uppercase variants of the keywords.

Figure 9.2 is an example of one of the rules found in `VBase`. Here, the rule matches the keyword 'signal' and ensures that at least one whitespace exists between this keyword and the next token/string that is matched.

```
Rule SIGNAL()  { return Sequence(IgnoreCase("signal"), W()); }
```

Figure 9.2: Implementation of rule used to match the keyword 'signal' and consume the necessary whitespace following the keyword

9.2 VHDL *Recognizer*

Write a recognizer using Parboiled for the VHDL grammar defined in Figure 9.1. The class you will be inserting your solution is `VRecognizer` in the package `ece351.vhdl`.

Running `TestVRecognizerAccept` and `TestVRecognizerReject` (found in `ece351.vhdl.test`) will test your VHDL recognizer against grammatical and ungrammatical inputs, respectively.

9.3 VHDL *Parser*

Write a parser using Parboiled for the VHDL grammar defined in Figure 9.1. The parser should return the corresponding AST (i.e., a `VProgram` object) of for any valid input program that the parser reads in.

The AST classes that are required to construct ASTs for VHDL programs are found in the packages `ece351.vhdl.ast` and `ece351.common.ast`. Note that assignment statements and expressions share the same AST classes used to create ASTs representing $\mathcal{F}$ programs.

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.

You can use `TestVParser` in the package `ece351.vhdl.test` to test out your parser.

9.4 VHDL → VHDL*: Desugaring*

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 refactored. For example, the expression $x \oplus y$ is equivalent to:

$$x \cdot \neg y + \neg x \cdot y \tag{9.1}$$

The output of the desugarer will be the transformed vhdl AST.

All of the code you will write for the desugarer will be in the `DeSugarer` class found in the package `ece351.vhdl`. `TestDeSugarer` in the package `ece351.vhdl.test` is available for you to test your vhdl desugarer.

9.5 *Define-Before-Use Checker [bonus]*

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 9.3, 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 9.3. 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`.

9.5.1 *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.

```
 1  entity four_bit_ripple_carry_adder is port (
 2      a0, b0, a1, b1, a2, b2, a3, b3, Cin : in bit;
 3      sum0, sum1, sum2, sum3,Cout, V: out bit
 4  );
 5  end four_bit_ripple_carry_adder;
 6
 7
 8  architecture fouradder_structure of four_bit_ripple_carry_adder is
 9       signal c0, c1, c2, c3, c4: bit;
10  begin
11      FA0 : entity work.full_adder port map(a0,b0,Cin,sum0,c1);
12      FA1 : entity work.full_adder port map(a1,b1,c1,sum1,c2);
13      FA2 : entity work.full_adder port map(a2,b2,c2,sum2,c3);
14      FA3 : entity work.full_adder port map(a3,b3,c3,sum3,c4);
15
16      V <= c xor c4;
17      Cin <= c4;
18  end fouradder_structure;
```

Figure 9.3: VHDL program used to illustrate signal declarations and the use of undefined signals in signal assignment statements.

Lab 10
VHDL → VHDL: *Elaboration*

In this lab, you will be writing a VHDL to VHDL transformer. The 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*.

Files:
ece351.vhdl.Elaborator.java
Tests:
ece351.vhdl.test.TestElaborator

10.1 Elaboration

The following sections describe the expected behaviour of the elaborator.

10.1.1 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.

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

Figure 10.1: vhdl program used to illustrate elaboration.

```
entity OR_gate_2 is port (
    x , y: in bit;
    F: out bit
);
end OR_gate_2;

architecture OR_gate_arch of OR_gate_2 is begin
    F <= x or y;
end OR_gate_arch;

entity four_port_OR_gate_2 is port (
    a,b,c,d : in bit;
    result : out bit
);
end four_port_OR_gate_2;

architecture four_port_structure of four_port_OR_gate_2 is
    signal e, f : bit;
begin
    OR1: entity work.OR_gate_2 port map(a,b,e);
    OR2: entity work.OR_gate_2 port map(c,d,f);
    result <= e or f;
end four_port_structure;
```

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).

Figure 10.2: Elaborated architecture body, `four_port_structure`.

```
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;
```

10.1.2 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

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

change the logical behaviour of the program. The result of elaborating `eight_port_OR_gate_2` is shown Figure 10.4.

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

```
        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;
```

Figure 10.4: Elaborated architecture body, `eight_port_structure`.

10.1.3 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.

10.2 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).

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.

11.2 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.

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

- **Hint:** Because the `ImmutableList` class does not offer methods to remove specific objects from the list, maintain a new ImmutableList in the splitting process to facilitate the process replacement that may occur.
- 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. The output $\mathcal{F}$ program should be built using the private variable `fprogram` defined in the class `ece351.vhdl.Synthesizer`.

Recall that the $\mathcal{F}$ program grammar is defined as shown in Figure 11.3.

```
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;
```

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

Program	→	Formula*
Fomula	→	Var '<=' Expr ';'
Expr	→	Term TermTail
TermTail	→	'+' Term TermTail | ϵ
Term	→	Factor FactorTail
FactorTail	→	'.' Factor FactorTail | ϵ
Factor	→	'!' Factor | '(' Expr ')' | Var | Constant
Constant	→	'0' | '1'
Var	→	id

Figure 11.3: LL(1) Grammar for $\mathcal{F}$. The use of the desugarer becomes apparent now; $\mathcal{F}$ only supports the 'and', 'or', and 'not' operators.

11.3.1 Expressions

Unlike the previous VHDL transforms, where all of the classes that you worked on extended the `PostOrderVExprVisitor` class, the synthesizer extends the `VExprVisitor` class. To generate $\mathcal{F}$ programs, what you will need to do is to visit all expressions in the AST in infix order.

In `Synthesizer.java`, write the code necessary to traverse through `Expr`s in infix order in the three `traverse` methods:

```
public Expr traverse(final NaryExpr e);
public Expr traverse(final BinaryExpr e);
public Expr traverse(final NotExpr e);
```

11.3.2 Signal Assignment Statements

Translating VHDL signal assignment statements into formulae in $\mathcal{F}$ is straightforward. The synthesizer will traverse through the signal assignment statements and make the necessary syntax changes to make the statement a valid $\mathcal{F}$ formula.

To reduce the likelihood of generating formulae with conflicting variable/signal names, prepend all signals with the name of the entity in which the signal is used/defined.

11.3.3 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, traverse through the signal assignment statements in the if- and else- bodies. Here, you will need to output formulae that reflects the logical behaviour of the if and else clause. Suppose that you encounter the following if/else statement[1]:

[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.

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

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

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

11.4 Example

11.4.1 Synthesizing Signal Assignment Statements

Consider the VHDL program shown in Figure 11.4. When the synthesizer processes this program[2], an $\mathcal{F}$ program consisting of a single formula (corresponding to) is generated (see Figure 11.5).

[2] after desugaring, elaborating, and splitting

Figure 11.4: 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 11.5: Synthesized output of the program in Figure 11.4.

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

11.4.2 Synthesizing If/Else Statements

Consider the VHDL program show in Figure 11.6. After performing all of the VHDL transformations that you have written, the synthesizer will generate the $\mathcal{F}$ program shown in Figure 11.7.

In Figure 11.7, observe that the synthesizer translates the if/else statement in Figure 11.6 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`.

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

```
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' ) );
```

Appendix A Academic Background

A.1 Mathematics & Logic

You are expected to have previously studied Boolean logic. You should be comfortable with Boolean formulæ and Venn diagrams. You may not remember the definitions of the following words, but you should have seen those definitions at some point in the past and be able to quickly understand the definitions when you see them again: commutativity, associativity, transitivity, reflexivity, absorption, De Morgan's Laws, *etc.*

Figure A.1 summarizes different notations for the three most common logical operators. We will use the hardware symbols on the board and in typeset formulæ; in ASCII text files we will use characters that are visually similar to these symbols or are commonly used for them. For your reference, Figure A.1 summarizes the alternative symbols that software engineers, logicians, and mathematicians tend to use for these operators.

Gate	**Name**	**HW Symbol**	**HW ASCII**	**SW Symbol**
AND	conjunction	$\cdot$	.	$\wedge$
OR	disjunction	$+$	+	$\vee$
NOT	negation	$\overline{x}$	!	$\neg$

Figure A.1: Logical operators and their symbols

A.2 Programming

- nested expressions / evaluation order
- recursive functions
- recursive structures
- classes + instances
- inheritance
  - extends
  - implements

  - instanceof
  - casting
- polymorphism
- fields are global to the methods of the class
- statics are global to the instances of the class
- Override@
- equals, hashCode, toString
- visibility: public / private / protected / default

A.3 Data Structures

Trees: traversal orders (preorder, postorder, levelorder, inorder)
Stacks: push, pop, peek, poke, swap

A.4 How programs execute

call stack: ece222, ece254
heap

A.5 VHDL

You should know the basics ... ECE124 ECE327

A.6 Notes from Class

Jan 10ish ...

```
- add Java/TXL explanation blurb in lab description
- don't need to know TXL
- specify output:
- stdout
- compute output name from input name
- expliclity specify output file name
- emphasize respect of the interfaces
- whitespace is insignificant
- don't worry about malformed inputs
- question about the input? run TXL program. if the TXL program processes it, so should you.
```

Jan 16

```
- wave files on classpath to be loaded
- instantiate W2SVG, rather than static version
- exam:
    - lexical vs dynamic scoping
    - name capture
    - call by name vs call by value
    - functional / imperative / declarative programming
- regular languages, pushdown automata, turing completeness
```

```
    - syntactic complexity vs expressive power: e.g., Java is
      context-free to parse, but is expressively turing complete
- finite state machines
    - reject when ...
    - accept when
    - recognize / match
    - ece124
- matching parentheses with a stack
- vhdl is context-sensitive?
- parsing context-sensitive languages is an AI issue?
- code review:
    - private fields
    - non-static fields
    - respect the interface
    - @Override
    - instantiating W2SVG
    - declaring vars within blocks
```

Jan 19

```
- parsing creates trees
- parsing vs lexing
- grammars have productions/rules
- your code should be structured around the grammar
- how does parboiled work?
- lab01:
    - students make bogus packages
    - students should write to stdout -- easier testing
    - some students submitted code that didn't compile
```

Appendix B
Computing Environment

You have three main options: remote log in to the course virtual machine (either by ssh or NX); run the course VM on your local computer; or install the course software on your own computer. In the past we have observed that some students have difficulty installing the software on their own computer, which is why we now offer the other two options.

B.1 Remote Login to ece351.uwaterloo.ca

In our experience these remote login options work best with a wired network connection. Wireless tends to have high latency and intermittent connectivity. Keep your expectations for wireless low and you may be pleasantly surprised.

B.1.1 PuTTY + Xming: Windows, on campus

If you are using a Windows machine, then you will need to download and install PuTTY [1] (Telnet/SSH client) and Xming [2] (X server for Windows). With PuTTY, simply put it in a convenient location.

Ensure that Xming is running when you start your SSH session. Open up PuTTY. In the PuTTY Configuration dialog, under Connection $\rightarrow$ SSH $\rightarrow$ X11, ensure that 'Enable X11 forwarding' is checked off. Go back to the 'Session' options and enter user@ece351.uwaterloo.ca as the host name. By default, the session options should be set to connect via SSH.

[1] http://www.chiark.greenend.org.uk/~sgtatham/putty/

[2] http://www.straightrunning.com/XmingNotes/

*B.1.2 ssh -X: *nix and Mac, on campus*

If you are using a Mac, then you will need to install X11, which is available on the Developer Tools CD that came with the computer, and may also be available on Apple's website[3]. Use the following command to SSH into ece351.uwaterloo.ca:

[3] As of Mac OS X 10.5 (Leopard), X11 is installed by default.

```
ssh -X user@ece351.uwaterloo.ca
```

To enable X11 forwarding over SSH, you may need to edit two SSH configuration files. Saving the changes to these files require sudo access.

The following lines should be present in the file ssh_config:

```
ForwardAgent yes
ForwardX11 yes
ForwardX11Trusted yes
```

If you just wish to configure SSH on a per-user basis, then you can add the three lines in the file ~/.ssh/config instead of ssh_config.

For Mac users, ssh_config reside in the directory /etc. For *nix users, these configuration files likely reside in the directory /etc/ssh. You may need to restart SSH for these changes to take effect.

B.1.3 NX: on or off campus

NX is a compressed, asynchronus version of the X11 protocol. It performs better over high latency network links. You can download the NX client software at no charge from the NoMachine corporation.[4]

[4] http://nomachine.com

B.2 Run the ECE351 Virtual Machine on your computer

You may get a copy of the ECE351 virtual machine (VM) from the lab instructor. This VM is derived from ece351.uwaterloo.ca. Then you can run the ECE351 environment on your local machine.

B.3 Configure your own computer

This option is not recommended for most students. You will need to install all of the tools listed in Appendix C, and possibly some others (the listing in Appendx C may not be complete). If you are on Windows you will need to install Cygwin[5]. If you are on Mac you may want to install the MacPorts[6] or Fink[7] package managers and use them to install some of the software.

[5] http://www.cygwin.com/

[6] http://www.macports.org/

[7] http://www.finkproject.org/

Appendix C
Tools

C.1 Command Line

ECE351 is a compilers course: it will require a fair amount of programming and some 'basic' computer skills, including:

- File extensions:
  - How to see the extension of a file in your file system explorer/navigator/browser.
  - How to change the extension of a file.

Some computers are configured so that the extension is hidden from the user.

- Many file types are just different types of text files. All of these files are edited with a text editor. This is different than the world of proprietary binary files where there is typically one (or perhaps more) program for each file extension: with text files we have many extensions for one program (your text editor).

For example: .java, .py, .txt, .tex, .html, .xml, .svg, *etc.*

C.1.1 Terminal Emulators

xterm: the original, configured via text files
xfce4-terminal: has tabs, GUI configuration options
copy and paste between terminal and text editor

C.1.2 Shells

BASH is the default on ece351.uwaterloo.ca, and so we'll assume that is what you are using. Alternatives include CSH and ZSH, which have slightly different syntax for certain things.
command completion: press tab
shell's search path
navigating the file system
basic commands: ls, cd, cp, mv, *etc.*

Don't use `cp` and `mv` in your working copy! Use `svn cp` and `svn mv` instead. Or, even easier, use Eclipse or TortoiseSVN.

C.1.3 *Piping and Redirection*

C.1.4 *Grep*

- . matches any character
- $*$ is zero or more
- $+$ is one or more
- ? is zero or one

C.1.5 *Diff*

-w

-u

C.2 *Text Editor*

Text editor:

Notepad++ seems to be popular with students running Windows. TextWrangler is a good Mac-only text editor that is available free of charge. The classic *nix text editors are Emacs and Vi.

- Know what a text editor is.
- Have a text editor installed on your computer.
- Know how to use that text editor.

Eclipse, Emacs, vi, pico, gedit, jedit,

C.3 *Subversion*

See https://ecesvn.uwaterloo.ca

C.4 *Eclipse*

Eclipse without subversion might copy the .svn files from src/ into bin/, which could lead to inadvertently deleting files from src/ on the repository.

Add the .svn folder to the Excluded list for your Source Folder.

This is happening because you checked out a project using another svn client other than eclipse and then imported the project in eclipse and you haven't notified eclipse that this is an svn project (that is, eclipse doesn't know it has to ignore the svn meta information).

In order to fix this properly, after you have imported the project in eclipse, have eclipse be 'aware' of the svn nature of the project. Do so by 1) select the project and to to the Team > Share section. 2) a dialog will appear asking weather this is a CVS or SVN project. Select the later. 3) It will prompt for the credentials, enter them. 4) When you're done, do a clean build. The problem goes away.

C.5 JUnit

C.6 Make

C.7 Graphviz

C.8 TXL

C.9 Parboiled

Appendix D
Grammar Design

- LALR
- LL
- GLR
- PEG

D.1 Designing a Grammar for $\mathcal{F}$

Is the grammar in Figure D.1 different from the grammar in Figure 3.1? What are the FIRST, FOLLOW, and PREDICT sets for the grammar in Figure D.1?

```
Value   := [0-1] / Var / '(' Expr ')'
Product := Value ('.' Value)*
Sum     := Product ('+' Product)*
Expr    := Sum

Indirect left recursion
Value   := [0-1]+ / '(' Expr ')'
Product := Expr ('.' Expr)*
Sum     := Expr ('+' Expr)*
Expr    := Product / Sum / Value
```

Figure D.1: Alternative Grammar for $\mathcal{F}$

Program	→	Formula*
Fomula	→	Var '=' Expr EOL
Expr	→	Term ('+' Term)*
Term	→	Factor ('.' Factor)*
Factor	→	'!' Factor | '(' Expr ')' | Var | Constant
Constant	→	'0' | '1'
Var	→	id

This grammar has some shortcomings: it does not generate `x = a + b + c`, and it does not encode the desired operator precedence.

Figure D.2: Another grammar for $\mathcal{F}$

Program	$\rightarrow$	Formula*
Formula	$\rightarrow$	Var '=' Expr
Expr	$\rightarrow$	Term
	|	Term BinaryOp Term
Term	$\rightarrow$	UnaryOp Term
	|	Constant
	|	Var
	|	'(' Expr ')'
UnaryOp	$\rightarrow$	'!'
BinaryOp	$\rightarrow$	'+' | '.'
Constant	$\rightarrow$	'0' | '1'
Var	$\rightarrow$	id

Figure D.3: Alternative definition for expr

Expr	$\rightarrow$	Constant
	|	UnaryOp Var
	|	Var BinaryOp Var
	|	'(' Expr ')'

What if we define expr like so:

a. Is the grammar LL(1)? In other words, are all FIRST sets disjoint? If any of the FIRST sets overlap with each other then the grammar will require more than one token of lookahead.

If the grammar is not LL(1) then you will likely choose to use a parsing tool rather than to write a recursive descent parser by hand.

b. Is the grammar left-recursive?
c. Is the grammar ambiguous?
d. Does the grammar preserve desired operator precedence?
e. Does the grammar generate all desired sentences?

D.2 Grammar Refactoring

D.2.1 Stratification

D.2.2 Removing Left-Recursion

D.2.3 Removing Ambiguity

Appendix E
Translator Design

E.1 Basic Design Patterns[1]

Where do these occur in the labs?

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

E.1.1 Singleton

E.1.2 Composite

E.1.3 Template Method

E.1.4 Interpreter

E.1.5 Visitor

E.1.6 Flyweight

E.1.7 Iterator

E.2 Write by hand vs. *Parser Generator* vs. *Library* vs. *Parser Combinator* vs. *PEG*

E.3 Event-Driven vs. *Representational*

event-driven:

- knows where it is in the grammar, but doesn't know what specific terminals it has seen in the past
- sequential access
- recognizer + processor
- smaller memory footprint

representational:

- larger memory footprint
- random access
- has entire tree in memory

E.4 Event-Driven

E.4.1 Push vs. *Pull*

The Streaming API for XML (StAX) is the new generation of XML APIs in Java. StAX is based on the so-called pull model in which an application queries the parser for the next parsing event, but never surrenders control to the parser during the process. Stated differently, StAX essentially turns the SAX processing model upside down. Instead of the parser controlling the application's flow, and the application reacting to parsing events, it is the application that controls the flow by pulling events from the parser.

This parsing model has several advantages over SAX. First, it often makes the application logic easier to understand given that it is the application and not the parser that is in control of the process (stated differently, the application does not get "pushed around" :). Second, if implemented correctly, there are a number of new optimizations that are possible when the application does not need to process the entire infoset. In particular, it is possible to lazily wait until the application requests a certain infoset item before it is actually constructed (a good example of this is lazy creation of Java strings).

Both StAX and SAX work in streaming fashion and allow only forward reading. However, the StAX model gives the application a lot more control: for example, applications have the option of pausing and resuming the parsing process, skipping over unneeded content, etc. For further information please refer to the Java Web Services Tutorial.

The StAX cursor model (XMLStreamReader) is the most efficient way to parse XML since it provide a natural interface by which the parser can compute values lazily. SJSXP takes full advantage of this by delaying the creation of certain objects until they are needed. SJSXP's XML token scanner is based on Xerces 2 but has been modified to accomodate the new pull model. The result is an implementation that is fully complaint with the XML specifications while at the same time offering very good performance.

http://sjsxp.java.net/

E.5 Representational

E.5.1 Visitor vs. *Interpreter*

E.5.2 Mutating vs. *Functional*

Appendix F
Designing a Circuit Optimizer

XOR: $A \oplus B = (A \cdot \bar{B}) + (\bar{A} \cdot B)$

Circuit Minimization

like Common Subexpression Elimination

Quine-McClusky / Karnaugh Maps

ESPRESSO[1]

Compact Boolean Circuits[2]

Boolean Expression Diagrams[3]

Reduced Boolean Circuits[4]

Logical vs Algebraic equivalence: if there are certain inputs for which we do not care about what the output is then there may be different circuits that compute equivalent values for all of the inputs we do care about (logically equivalent circuits) but produce different outputs for the inputs we do not care about (*i.e.*, not algebraically equivalent).

Optimizing multiple output pins simultaneously: search for common product terms (*cf.*, our lab on common subexpression elimination).

[1] http://en.wikipedia.org/wiki/Espresso_heuristic_logic_minimizer

[2] E. Torlak and D. Jackson. Kodkod: A relational model finder. In O. Grumberg and M. Huth, editors, *Proc.13th TACAS*, volume 4424 of *LNCS*, pages 632–647, Braga, Portugal, Mar. 2007. Springer-Verlag; and E. Torlak. *A Constraint Solver for Software Engineering: Finding Models and Cores of Large Relational Specifications*. PhD thesis, MIT, 2009

[3] H. Anderson and H. Hulgaard. Boolean expression diagrams. In *Proc.12th IEEE Symposium on Logic in Computer Science (LICS)*, Warsaw, Poland, June 1997

[4] P. Abdulla, P. Bjesse, and N. Eén. Symbolic reachability analysis based on SAT-solvers. In S. Graf and M. I. Schwartzbach, editors, *Proc.6th TACAS*, volume 1785 of *LNCS*, pages 411–425, Berlin, Germany, Mar. 2000. Springer-Verlag

F.1 Term Rewriting: A Syntactic Approach to Optimization

Although the language is very convenient to represent the structure of a circuit, the constitution of a circuit may be very complicated sometimes. Programmers are often asked to improve the design of circuits.

For instance, designers may be asked to design a circuit that has minimum latency. To realize the function, the circuit must have as less level gates as possible; therefore, it is usually designed as a and-or circuit. The way to transform the circuit to that form in formula language f is “multiplying” out the inputs together. However, a more efficient way is to use the program of optimization.

The line of formula shown in the previous sub-section is a good example. The key point in the program to optimize the circuit is developing a rule distributing all the inputs which is called distributor. The details of the program is shown in the next figure.

Figure F.1: Grammar for distributor in TXL.

```
rule distributor1
        replace [expression]
                e [expression]
        deconstruct e
                (r1 [relation] or r2 [relation]) and r3 [relation]
        by
                (r1 and r3) or (r2 and r3)
end rule

rule distributor2
        replace [expression]
                e [expression]
        deconstruct e
                r3 [relation] and (r1 [relation] or r2 [relation])
        by
                (r3 and r1) or (r3 and r2)
end rule
```

In figure F.1, there are two distributors in the program since the relation outside a bracket can be either in front of the bracket or after the bracket. Both rules work on transforming the product of sum into the sum of product. Since the rule in TXL will run recursively, the program will not stop until the program find the last matching input. Therefore, the output after applying the rules is

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

which is the correct answer.

Moreover, the program not only need distributors, but also be supposed to do absorption in order to delete some gates which are redundant. For instance, a expression indicated as "a and (a or b)" can be simplified as "a". The solution in figure F.1 demonstrates a way to finish the absorber in TXL. As distributor requires two rules, absorber actually have to have four rules because the position of the relation out of bracket is indeterminate and the position of the factor which is going to be eliminated in the bracket is uncertain. Finally, the rules will keep being applied until the input file cannot be simplified anymore for the same reason as distributors.

After finishing the distributor and absorber, programs are facing a problem: which rule should be applied firstly? Although TXL is relatively smarter than many other parsing tools like ANTLR since it can be executed recursively and has a back-tracking function, the order of rules being applied does matter in some cases. Considering the following example.

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

If the distributor rules are applied firstly, the or-gate out of brackets will be removed and will produce the output as:

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

However, this output is obviously not minimal. If the relation (A and B) is replaced by a factor H, and the relation (C and (D or E)) is substituted by a factor I, the original function will become H and (H or I), which can be simplified as H. H can be replaced back to (A and B) finally. The reasons why the output is different with the desired output are that the expression in the second bracket becomes "((A and B) and (C and D)) or ((A and B) and (C and E))" and the TXL program does not have a absorber to deal with such complicated expression.

On the contrary, the formula can generates perfect output if the rules of absorber are pulled ahead of the rules of distributor.

In conclusion, the order of application of rules is very important. Wrong sequence may lead people to a unexpected output. Therefore, programmers have to choose the order that they apply rules wisely.

Appendix G
Bibliography

[1] P. Abdulla, P. Bjesse, and N. Eén. Symbolic reachability analysis based on SAT-solvers. In S. Graf and M. I. Schwartzbach, editors, *Proc.6th TACAS*, volume 1785 of *LNCS*, pages 411–425, Berlin, Germany, Mar. 2000. Springer-Verlag.

[2] H. Anderson and H. Hulgaard. Boolean expression diagrams. In *Proc.12th IEEE Symposium on Logic in Computer Science (LICS)*, Warsaw, Poland, June 1997.

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

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

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

[6] B. Eckel. *Thinking in Java*. Prentice-Hall, 2002.

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

[8] W. L. Hürsch. Should superclasses be abstract? In M. Tokoro and R. Pareschi, editors, *Proc.9th ECOOP*, volume 821 of *LNCS*, pages 12 – 31, Bologna, Italy, July 1994. Springer-Verlag.

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

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

[11] E. Torlak. *A Constraint Solver for Software Engineering: Finding Models and Cores of Large Relational Specifications*. PhD thesis, MIT, 2009.

[12] E. Torlak and D. Jackson. Kodkod: A relational model finder. In O. Grumberg and M. Huth, editors, *Proc.13th TACAS*, volume 4424 of *LNCS*, pages 632–647, Braga, Portugal, Mar. 2007. Springer-Verlag.