DEREK RAYSIDE & ECE351 STAFF

ECE351 COURSE NOTES

UNIVERSITY OF WATERLOO

Copyright © 2018 Derek Rayside & ECE351 Staff
Compiled December 12, 2018

ACKNOWLEDGEMENTS:

- Jon Eyolfson [TA S2012; Lab Instructor W2013 & S2013]
- Students of W2012: David Choi, Michael Lin, Aman Muthrej, Atulan Zaman
- Student Parul Arora from W2018.

Licensed under Creative Commons Attribution-ShareAlike (CC BY-SA) version 2.5 or greater.
`http://creativecommons.org/licenses/by-sa/2.5/ca/`
`http://creativecommons.org/licenses/by-sa/3.0/`

Contents

0 Engineering a Compiler 11
0.1 What is a Compiler? 11
0.2 Stages of Compilation 12
0.3 Engineering Design 13

1 Program Understanding 15
1.1 Object Diagrams 16
1.1.1 Learning Object Diagrams from Superheros 16
1.1.2 Harry Potter's Origin Story 17
1.1.3 Superman's Origin Story 18
1.1.4 Aliasing 19
1.1.5 Mutation and Aliasing with JDK LinkedList 21
1.1.6 Mutation and Aliasing with Parboiled ImmutableList 21
1.1.7 The keyword 'final' prevents re-assignment 23
1.2 Algorithmic Complexity 24
1.2.1 NP-Completeness and Boolean SATisfiability 24
1.2.2 Undecidability and The Halting Problem 25
1.2.3 Easy, Hard, and Impossible 26
1.3 Four Variants of Linear Search 28
1.4 Programming 'Paradigms' 34

2 Regular Languages & Finite Automata 37
Tiger: §2.3–5
2.1 Kinds of Machines and Computational Power 37
2.1.1 Turing Machines and Programming Languages 38

2.2 *Regular Expressions* 38
2.3 *Finite State Machines* 40
2.4 *Regex* → NFA 41
2.5 NFA → DFA 42
2.6 DFA *Minimization* 42
2.7 *Example regex* → NFA → DFA → *minimized* DFA 43
2.8 DFA *Minimization with an Explicit Error State* 44
2.9 *Another Example* 45
2.9.1 *Regex → NFA* 45
2.9.2 *NFA → DFA* 45
2.9.3 *DFA → minimized DFA* 45
2.10 *Finite Automata in* ECE351 VS. ECE327 46
2.11 *Additional Exercises* 47

3 *Classifying Grammars by Complexity* 57
3.1 *Always choose the simplest possible complexity class* 59
3.1.1 *Refactoring and the Equivalence of Grammars* 59
3.2 *Is this grammar context-free?* 59
3.3 *Is this grammar regular?* 60
3.3.1 *Common cases outside the regular class* 60
3.3.2 *Proving a grammar is regular* 60
3.4 *Is this grammar ambiguous?* 61
3.4.1 *Removing ambiguity using precedence* 62
3.4.2 *Removing ambiguity using associativity* 63
3.5 *Is this grammar* LL(1)*? Simple tests* 66
3.5.1 *Remove common prefixes with left-factoring* 66
3.5.2 *Remove left recursion* 66
3.6 *Is this grammar* LL(1)*? Full test* 67
3.6.1 *Convert* EBNF *to* BNF 68
3.6.2 *Which nonterminals are nullable?* 69
3.6.3 FIRST *sets* 69
3.6.4 FOLLOW *sets* 70
3.6.5 PREDICT *sets* 71

Tiger: p.25

Tiger: p.27

Tiger: not covered.

Tiger: p.84

3.7 *Is this a PEG? (Parsing Expression Grammar)* 72
3.8 *Grammar Design* 72
3.9 *Additional Exercises* 76

4 *Midterm — What you should know so far* 85

5 *Case Studies* 87
5.1 *Git* 87
5.2 LLVM 89
5.3 *Java Virtual Machine & Common Language Runtime* 91
5.3.1 *Comparison with VMWare, VirtualBox,* etc. 92
5.3.2 *Structure of the Call Stack* 93
5.3.3 *Object Header and Type Information Block* 94
5.4 *Cfront: Translating C++ to C* 96
5.4.1 *Optimizing Polymorphic Calls* 101
Tiger: §14.7
5.4.2 *Why have polymorphism as a language feature?* 101
5.4.3 *Subtype Polymorphism and Parametric Polymorphism* 101

6 *Optimization* 105
Tiger: §10.0–1, §17.0, §17.2–3
6.1 *A Learning Progression Approach to Dataflow Analysis* 105
6.2 *Optimization by Intuition* 106
6.3 *Optimization Step By Step* 107
6.4 *Convert to Three-Address Form* 107
6.5 *Available Expressions Dataflow Analysis on Straightline Code* 109
6.6 *Dataflow Analysis on Programs With Loops & Branches* 111
6.6.1 *Iteration to a Fixed Point* 111
6.7 *Available Expressions Dataflow Analysis (with loops)* 114
6.8 *Reaching Definitions Dataflow Analysis (with loops)* 119
6.9 *Duality of Available Expressions and Reaching Definitions* 121
6.10 *Summary* 122

7 *Storage Management* 123

7.1 *Register Allocation* 123

Tiger: §11.0, 21.0

7.1.1 *Liveness Analysis* 125

7.1.2 *Interference Graph Colouring* 126

7.2 *Garbage Collection* 127

Tiger: §13.0–4, §13.7.1

7.3 *Three options for cleanup* 128

7.4 *Reference Counting* 129

7.5 *Mark & Sweep* 129

7.6 *Semi-Space Copying Collection* 130

7.7 *Generational Collection* 130

7.8 *Discussion* 130

7.9 *Object Allocation with Free Lists* 131

7.10 *Fast Object Allocation* 131

7.11 *DieHard: Probabilistic Memory Safety for C* 132

7.12 *Memory Safety and Language Selection* 133

B *Bibliography* 137

J *Jokes on Engineering Practice* vs. *Theory* 139

J.1 *A theoreticians's salary* 139

J.2 *British* vs. *French Engineers* 140

J.3 *Engineer's Induction* 140

J.4 *Close enough for practical purposes* 142

List of Figures

1 A compiler transforms a string to a tree to another string 11
2 Stages of compilation (simplified) 12
3 Stages of compilation (typical C compiler) 12

1.1 Aliasing in *The Brothers Karamazov* 19
1.2 Aliasing and mutation in *Where The Wild Things Are* 20
1.3 Mutation and aliasing with JDK LinkedList 21
1.4 Mutation and aliasing with Parboiled ImmutableList 22
1.5 Correct usage of the 'final' keyword 23
1.6 Incorrect usage of the 'final' keyword 23
1.7 Summary of some big-O related terminology 24
1.8 Truth table for $x \cdot y \cdot z$ 25
1.9 Four different implementations of linear search. 29
1.10 Object diagram for iterative array variant of linear search 30
1.11 Object diagram for recursive array variant of linear search 31
1.12 Object diagram for iterative linked-list variant of linear search 32
1.13 Object diagram for recursive linked-list variant of linear search 33
1.14 Programming paradigms 35

2.1 Terminology for FSA/FSM/*etc.* 37
2.2 Grep is a compiler 39
2.3 Grep is an interpreter 39
2.4 Grep is an interpreter 39
2.5 Rules for converting regex to NFA [Tiger f2.6] 41
2.6 Rules for converting regex to NFA [PLP f2.7] 41
2.7 Example of converting regex to NFA [PLP f2.8]. The regex is: $d^*(\ .d \mid d.\ )\ d^*$ 43
2.8 Example of NFA to DFA conversion and DFA minimization [PLP f2.9 & f2.10] 43
2.9 An NFA and corresponding non-minimal DFA for the regular expression f?g*, as well as a bogus minimized DFA. 44
2.10 DFA with an explicit error state X. 44
2.11 Minimizing a DFA with an explicit error state. 44
2.12 Comparison of finite state machine notations in ECE351 and in ECE327 46

2.13 A simple ECE327 machine translated into an equivalent ECE351 machine 46

3.1 Chomsky hierarchy of grammatical complexity 57
3.2 Kinds of grammars and the machines needed to parse them 57
3.3 Language notation examples 58
3.4 Context-sensitive grammar for $\{a^n b^n c^n \mid n \geq 1\}$ 59
3.5 An EBNF grammar for $\mathcal{F}$, from LAB4 67
3.6 BNF version of $\mathcal{F}$ grammar from Figure 3.5. We've converted both the Kleene stars ('*') and the alternative bars ('|'). Converting the stars resulted in the introduction of three new nonterminals: FList, TermTail, and FactorTail. Converting the '|' just results in multiple lines with the same left-hand side (LHS). 68

5.1 Flipbook of skeleton Git repository 87
5.2 Flipbook of student Git repository 88
5.3 General structure of a simple compiler 89
5.4 General structure of a retargetable compiler 89
5.5 Structure of LLVM 89
5.6 Link-time optimization with LLVM 90
5.7 Virtual Machine 91
5.8 Structure of call stack (PLP f8.10) 93
5.9 Visualization of call stack and heap from `PythonTutor.com` 93
5.10 (PLP f9.3) 95
5.11 (PLP f9.4) 95
5.12 (Tiger f14.3) 95
5.13 Simple C++ to C translation 97
5.14 Simple C++ to C translation with VTables and with switch statements 98
5.15 C++ to C translation using VTables, with a polymorphic call 99
5.16 C++ to C translation using switch statements 100

6.1 Hasse diagrams for domains of powersets of two and three elements. Images from Wikipedia, under the GNU Free Documentation License. 112
6.2 Example flow graph 114
6.3 An example illustrating the difference between Available Expressions and Reaching Definitions. 121

7.1 Liveness analysis example 125
7.2 Interference graph example 126
7.3 Garbage collection in action. 127

Todo list

See `http://www.jflap.org/` for software for converting a DFA to an equivalent regular grammar . . . 60
Incorporate material from `http://theory.stanford.edu/\protect\unhbox\voidb@x\penalty\@M\{}amitp/yapps/yapps-doc/node3.html` . . . 63
Piazza post from Ryan . . . 70
Student confusion about why this production doesn't add (see suppressed text) . . . 70
gcc, Eclipse, edg, gdb, ghc, HipHop, v8, XUL, CLR, WAM . . . 103
See the Tiger §13 excerpt in the ECE351 notes repo. . . . 129
See the Tiger §13 excerpt in the ECE351 notes repo. . . . 130
See the Tiger §13 excerpt in the ECE351 notes repo. . . . 130
See the Tiger §13 excerpt in the ECE351 notes repo. . . . 131
See the Tiger §13 excerpt in the ECE351 notes repo. . . . 131

Engineering a Compiler

0.1 What is a Compiler?

What is a compiler? There are several common answers, such as:

- a program that transforms source code to binary code
- a program that transforms byte code to binary code
- a program that transforms source code to source code
- a program that transforms strings into trees, then back into strings

Consider the small program x=2+3. Figure 1 depicts this program as an input string, a tree (an *abstract syntax tree*, or AST), and finally as some resulting assembly code.

Figure 1: A compiler transforms a string to a tree to another string

source → parsing → AST → codegen → assembly

x=2+3

=
x
+
2
3

```
mov eax, 2
mov ebx, 3
add eax, ebx
push eax
```

0.2 *Stages of Compilation*

Compilers typically have a linear organization, with each stage producing data for the next stage. Figure 0.2 and Figure 0.2 depict some common stages as edges in a graph, where the nodes represent the kinds of data that are input and output for each stage (arrow). Figure 0.2 is a simplified view of compilation, whereas Figure 0.2 shows the stages in a typical C compiler. Modern Just-In-Time (JIT) compilers have some other stages, which we will discuss later in the term.

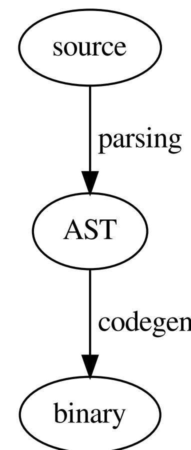


Figure 2: Stages of compilation (simplified)

SIMPLIFIED VIEW: COMPILING = PARSING + CODEGEN. At this high level, parsing transforms the input source *string* into a *tree* (the *abstract syntax tree*, or AST); and code generation transforms the AST into the output string (perhaps a binary string of bits).

At this level we can see compilers as a metaphor for any program that reads structured text input, analyzes and transforms the input, and finally produces output. A surprisingly wide variety of programs can be thought of in this way, and techniques from compilers can be helpful in designing and implementing such programs.

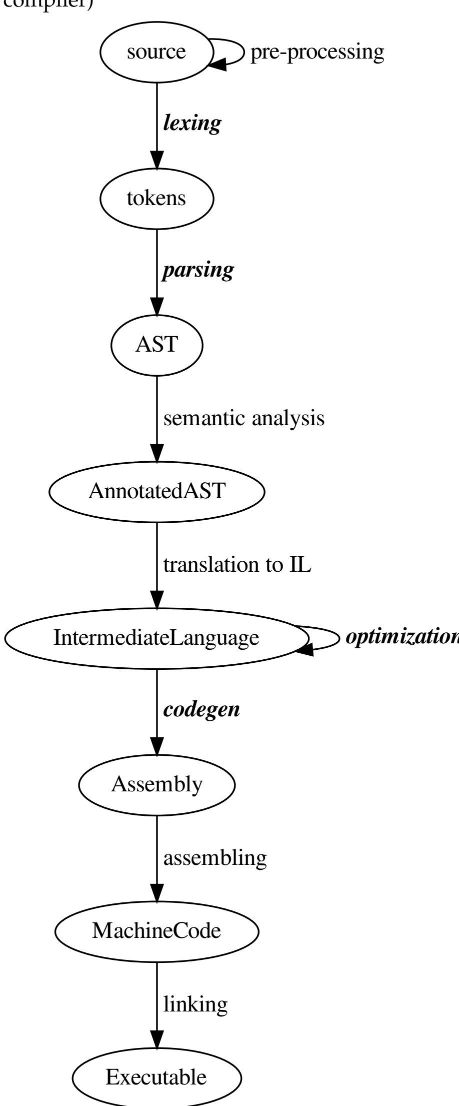


Figure 3: Stages of compilation (typical C compiler)

EXPANDED VIEW. A typical C compiler expands our simplified view with additional stages, as depicted in Figure 0.2. The stages that we will focus on in ECE351 are highlighted in the figure:

a. *Lexing:* chunk the input into a list of tokens (*i.e.*, words). Typically removes whitespace. Also known as *scanning* or *tokenizing*.
   - LAB1
   - §2

b. *Parsing:* build an AST from the sequence of tokens.
   - LAB1
   - LAB3
   - LAB5
   - LAB6
   - LAB9
   - §2
   - §3

c. *Optimization:*
   - LAB4: term-rewriting
   - §6 data-flow analysis based optimizations

d. *CodeGen:* we will study generating a variety of output forms, including some assembly codes.
   - LAB11?: JVM assembly
   - LAB12?: x86 assembly
   - §7: register allocation + object allocation
   - §**??**: dynamic dispatch

0.3 Engineering Design

ECE351 is an engineering design course. In engineering design we are concerned with questions such as the following:

- Will technique T work for problem P?
- How expensive is technique T?
- What kind of machine is required to solve problem P?
- What is easy? What is hard? What is impossible?
- When to use approximation?
- Making tradeoffs between available resources.

Throughout ECE351 we will consider these questions in a variety of contexts. As compiler engineers, the main resource trade-offs that we are concerned with are:

Consider the programming language feature of automatic array bounds checking. This costs the compiler engineer some time and cognition, makes the compiler run a little longer and use a bit more space, makes the resulting executable program a bit slower and require a bit more space (to remember the runtime size of every array) — but, it saves the programmer some time and cognition, and it saves the user from security holes (a *buffer overlow* attack is only possible against programs written in languages without automatic array bounds checking).

$$\textit{Resources} \begin{cases} \textit{engineer} & \textit{time} \;\&\; \textit{cognition} \\ \textit{compiler} & \textit{time} \;\&\; \textit{space} \\ \textit{executable} & \textit{time} \;\&\; \textit{space} \\ \textit{programmer} & \textit{time} \;\&\; \textit{cognition} \\ \textit{user} & \textit{time} \;\&\; \textit{security} \end{cases}$$

By *engineer* here we mean the person writing the compiler. By *executable* we mean the compiled program. By *programmer* we mean the person who will use the compiler. By *end user* we mean the person who will run the program written by the programmer and compiled by the compiler.

Engineering design decisions about resource tradeoffs are informed by both theoretical understanding and empirical experience. Compilers are a fascinating and industrially relevant area with both deep theory and significant empirical experience to draw on.

Chapter 1
Program Understanding

If we understand a program, then we know some things about it, including:

- Inputs
- Outputs (including errors)
- Static Structure:
  - data structures
  - class diagrams
  - design patterns
- Dynamic Execution:
  - control structures / call graph
  - pointers and aliasing
  - object diagrams (visualizations of program execution)
  - mutation (what data changes during execution)
  - invariants (data relationships that are always true)
- Algorithms:
  - runtime (both asymptotic/big-O and constant factors)
  - space usage (both asymptotic/big-O and constant factors)
  - termination conditions (when will the program end?)
- Programming paradigm/style

Some of these topics you will have been introduced to previously; some of these topics might be new to you. Understanding programs to this kind of depth is important for writing compilers, from two perspectives: first, compilers typically analyze the programs they are compiling in some of these ways; second, a compiler is typically a large and sophisticated program in itself, and it is very helpful to understand the programs that you are writing.

This chapter will introduce some of these topics, and others will be discussed as the course progresses.

Topics that will be discussed later in the course include design patterns, mutation, and invariants.

1.1 *Object Diagrams*

In ECE351 we will use *object diagrams* to visualize the state of an executing program. These diagrams will depict the objects, stack frames, heap, and pointers at a specific time point during the program execution. We will use the following notation:

Icon	***Description***
PC: 3	Program Counter (PC; the current line number in the execution).
main() / var1 / var2	Stack frame, with slots for variables. There might be a special frame for statics / globals.
Object1	An object (in the heap), named by it's class name plus a unique index number.
Object1 —field→ Object2	A field from Object1 referencing Object2 (label is optional).
main() / var1 - - → Object1	A variable in a stack frame referring/pointing to an object in the heap.

These ECE351 object diagrams are inspired by the *box-and-pointer* diagrams from MIT's classic textbook *The Structure and Interpretation of Computer Programs*, which is available online under a Creative Commons license.

The name 'object diagram' has also been used in the UML community and in other contexts. UML object diagrams are similar to ECE351 object diagrams, but UML object diagrams just depict the heap, whereas ECE351 object diagrams additionally include the call stack.

`PythonTutor.com`, by Prof Philip Guo in the Cognitive Science Department at the University of California San Diego, draws these kinds of diagrams for your program as it executes. `PythonTutor.com` supports other languages, including Java. You can run the example programs from this chapter in `PythonTutor.com` to see how it visualizes their execution.

We consider that, conceptually, all objects are allocated on the heap. Our view is that the compiler (not the programmer) will control whether allocation happens on the heap or on the stack. C++ is just about the only language where the programmer has control over this decision: in almost all other languages it is under the compiler's control.

1.1.1 *Learning Object Diagrams from Superheros*

Super-hero origin stories often follow a predictable pattern:

a. the hero is young, and perhaps special, but without superpowers
b. tragedy befalls the hero's family (often in the form of a villain)
c. the hero acquires superpowers, and perhaps an alias

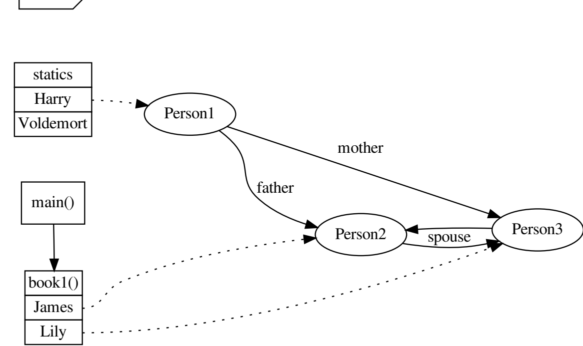
These kinds of patterns in literature and mythology are well-studied: *e.g.*, Joseph Campbell's classic book *The Hero With a Thousand Faces*.

We can write programs about superhero origin stories. These programs have no computational purpose: they have no input, no output, and no algorithm. Their purpose is pedagogigcal: hopefully anthropomorphizing program state and its transformations will help us learn how to conceptualize program execution more generally.

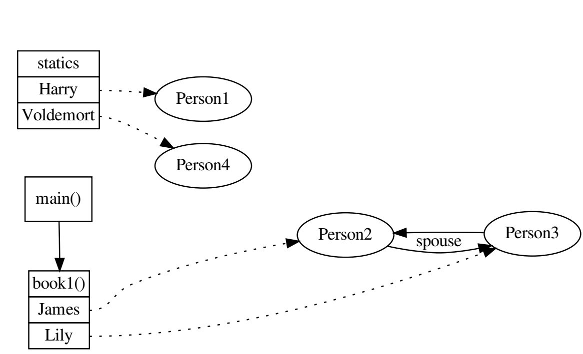
1.1.2 *Harry Potter's Origin Story*

```
1  class HarryPotterOriginStory {
2      static Person Harry;
3      static Person Voldemort;
4
5      public static void main(String[] args) {
6          book1();
7      }
8      static void book1() {
9          Harry = new Person();
10         Person James = new Person();
11         Person Lily = new Person();
12         Harry.father = James;
13         Harry.mother = Lily;
14         James.spouse = Lily;
15         Lily.spouse = James;
16         // object diagram 1
17         Voldemort = new Person();
18         Voldemort.attack(Harry);
19         // object diagram 2
20     }
21 }
22 class Person {
23     Person mother;
24     Person father;
25     Person spouse;
26     void attack(Person p) {
27         p.mother = null;
28         p.father = null;
29     }
30 }
```

The Potter Family Object Diagram



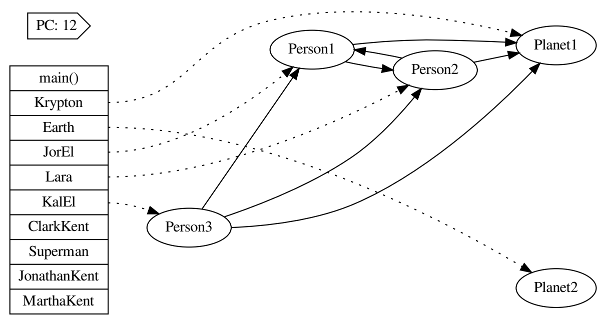
After the Attack



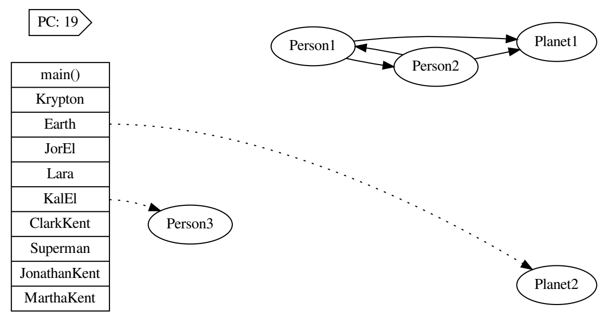
1.1.3 *Superman's Origin Story*

```
public class SupermanOriginStory {
    public static void main(String[] args) {
        // the universe
        Planet Krypton = new Planet();
        Planet Earth = new Planet();
        // parents
        Person JorEl = new Person(null,null,null,Krypton);
        Person Lara = new Person(null,null,JorEl,Krypton);
        JorEl.spouse = Lara;
        // KalEl is born
        Person KalEl = new Person(JorEl,Lara,null,Krypton);
        // object diagram 1
        // Planet Krypton explodes
        // JorEl and Lara launch KalEl towards earth
        Krypton = null; JorEl = null; Lara = null;
        KalEl.father = null;
        KalEl.mother = null;
        KalEl.planet = null;
        // object diagram 2
        // KalEl lands on earth
        KalEl.planet = Earth;
        // KalEl is adopted by the Kent family
        Person JonathanKent =
            new Person(null,null,null,Earth);
        Person MarthaKent =
            new Person(null,null,JonathanKent,Earth);
        JonathanKent.spouse = MarthaKent;
        Person ClarkKent = KalEl;
        ClarkKent.father = JonathanKent;
        ClarkKent.mother = MarthaKent;
        KalEl = null;
        // object diagram 3
        // Clark Kent creates his Superman alias
        Person SuperMan = ClarkKent;
        // object diagram 4
    }
}
class Planet {}
class Person {
    Person father, mother, spouse;
    Planet planet;
    Person(Person f, Person m, Person s, Planet p) {
        father = f; mother=m; spouse=s; planet=p;
    }
}
```

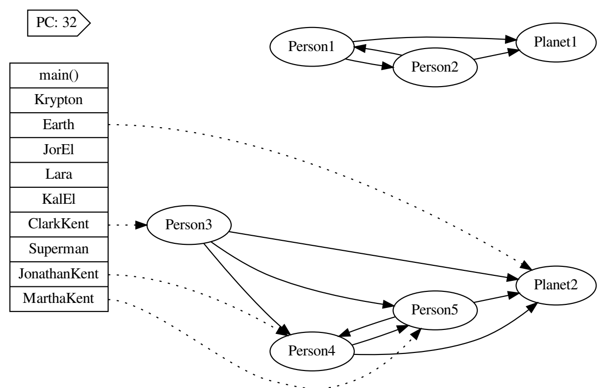
The El Family Object Diagram



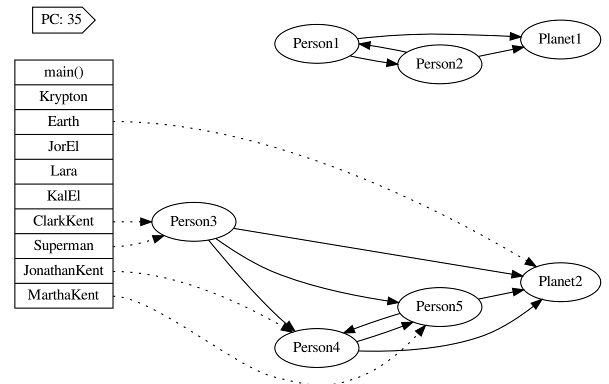
Planet Krypton Explodes



Kal-El is adopted by the Kent family



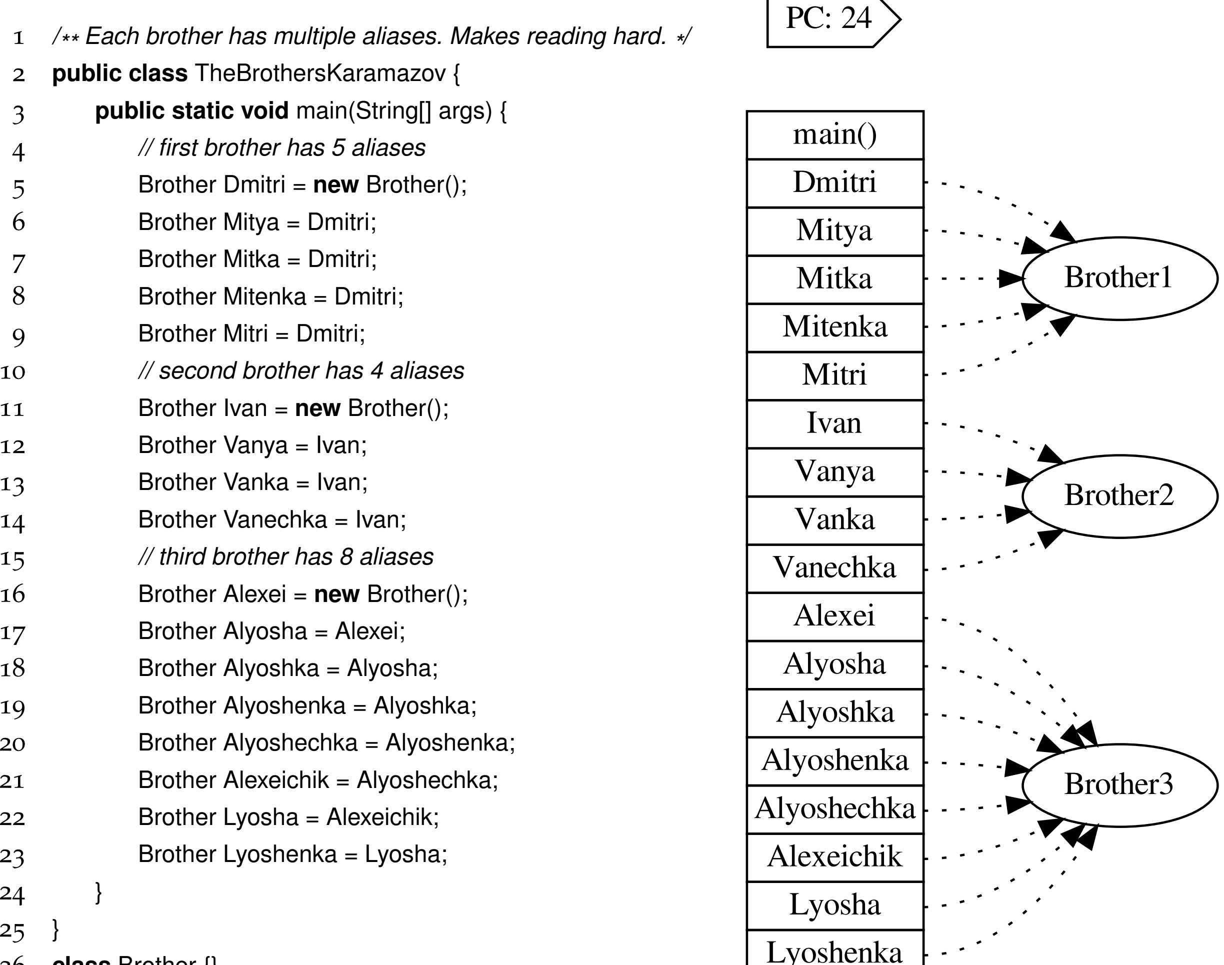
Clark Kent creates Superman alias



1.1.4 Aliasing

Aliasing is when multiple variables all refer to the same object. We saw this above in Superman's origin story: the three variables KalEl, ClarkKent, and SuperMan all refer to the same Person3 object. The situation is even more extreme in Fyodor Dostoevsky's classic novel *The Brothers Karamazov*, where each brother has multiple aliases (as was common in nineteenth-century Russia, where the novel is set).

Aliasing can make fiction more colourful, but can make programs harder to read and understand.



```
1  /** Each brother has multiple aliases. Makes reading hard. */
2  public class TheBrothersKaramazov {
3      public static void main(String[] args) {
4          // first brother has 5 aliases
5          Brother Dmitri = new Brother();
6          Brother Mitya = Dmitri;
7          Brother Mitka = Dmitri;
8          Brother Mitenka = Dmitri;
9          Brother Mitri = Dmitri;
10         // second brother has 4 aliases
11         Brother Ivan = new Brother();
12         Brother Vanya = Ivan;
13         Brother Vanka = Ivan;
14         Brother Vanechka = Ivan;
15         // third brother has 8 aliases
16         Brother Alexei = new Brother();
17         Brother Alyosha = Alexei;
18         Brother Alyoshka = Alyosha;
19         Brother Alyoshenka = Alyoshka;
20         Brother Alyoshechka = Alyoshenka;
21         Brother Alexeichik = Alyoshechka;
22         Brother Lyosha = Alexeichik;
23         Brother Lyoshenka = Lyosha;
24     }
25 }
26 class Brother {}
```


Figure 1.1: Aliasing in *The Brothers Karamazov* (novel by Fyodor Dostoevsky)

Aliasing occurs all the time in regular programs. The example above of *The Brothers Karamazov* was intentionally gratuitious. The next story, *Where The Wild Things Are* by Maurice Sendak, is a more realistic program, where we see that aliasing inevitably occurs due to procedure calls, and also sometimes is used to make the program easier to understand.

```
1  public class WhereTheWildThingsAre {
2      public static void main(String[] args) {
3          Person Mother = new Person(false, false);
4          Person Max = new Person(true, false);
5          Person MaxInWolfSuit = Max;
6          Mother.say("Wild thing!");
7          MaxInWolfSuit.say("I'll eat you up!");
8          sailAway(MaxInWolfSuit);
9          Max.eatSupper(); // and it was still hot
10         Max.isLonely = false;
11     }
12     static void sailAway(Person p) {
13         p.isLonely = true; // who is p?
14         WildThing[] wildthings = new WildThing[]{
15             new WildThing(), new WildThing(), new WildThing()};
16         for (WildThing w : wildthings) {
17             w.roar(); w.gnash(); w.claw(); }
18         if (p.canStareWithoutBlinking) {
19             Person King = p;
20             King.say("be still!");
21             King.say("let the wild rumpus start!");
22             King.rumpus();
23             for (WildThing w : wildthings) {
24                 w.rumpus(); }
25             King.say("now stop!");
26             if (King.isLonely) {
27                 wildthings[0].say("Oh please don't go!");
28                 wildthings[1].say("We'll eat you up!");
29                 wildthings[2].say("We love you so!");
30                 King.say("No!");
31                 for (WildThing w : wildthings) {
32                     w.roar(); w.gnash(); w.claw(); }
33                 return;
34             } else { sailAway(King); }
35         }
36     }
37 }
38 class Person {
39     boolean canStareWithoutBlinking, isLonely;
40     Person(boolean blink, boolean lonely) {
41         canStareWithoutBlinking = blink;
42         isLonely = lonely; }
43     void rumpus() {}; void eatSupper() {};
44     void say(String s) {System.out.println(s);}
45 }
46 class WildThing {
47     void roar() {}; void gnash() {}; void claw() {}; void rumpus() {};
48     void say(String s) {System.out.println(s);}
49 }
```

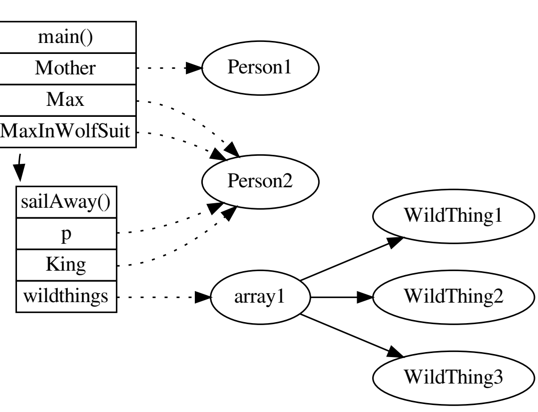


Figure 1.2: Aliasing and mutation in *Where The Wild Things Are* (story by Maurice Sendak)

Aliasing occurs in this program:

- The variable p illustrates that procedure parameters are unavoidable aliases.
- The variables MaxInWolfSuit and King are aliases that are added to the program to aid understanding.

Mutation occurs in this program:

- Line 13 mutates p.isLonely:
  - Who is p? Which object does p refer to?
- Line 10 mutates Max.isLonely:
  - The story reader knows this matches line 13.
  - Hard for the **compiler** to know. Must perform an *inter-procedural pointer analysis*, which is essentially constructing our object diagram (but for every possible program point, and every possible input).
  - Even harder for a **programmer** to find these matches in large programs that are not based on famous fictions.
  - Be careful about mutation and aliasing!

1.1.5 Mutation and Aliasing with JDK LinkedList

The standard data structures that ship with the Java Development Kit (JDK) are *mutable*, as illustrated in Figure 1.3. As we saw above, when mutation is combined with aliasing it can make the program harder to understand. In Figure 1.3, the variables list and alias both refer to the same object (LinkedList1), so lines 8 and 9 both mutate that same object, resulting in the output [hello, world].

Figure 1.3: Mutation and aliasing with JDK LinkedList

```
1  import java.util.List;
2  import java.util.LinkedList;
3
4  public class MutableListExample {
5      public static void main(String[] args) {
6          List list = new LinkedList();
7          List alias = list;
8          list.add("hello");
9          alias.add("world");
10         System.out.println(list); // ???
11     }
12 }
```


1.1.6 Mutation and Aliasing with Parboiled ImmutableList

Because procedure calls introduce necessary aliases, and because compilers are typically large programs with many procedure calls, some compiler engineers make the design choice to use *immutable* data structures: the objective is to make the compiler code easier to reason about, to reduce the introduction of bugs, and to make debugging easier. A trade-off is that adding new objects into an immutable data structure can feel different for the programmer.

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

In the ECE351 labs we will eventually use a parser generator named *Parboiled*. Parboiled comes with, and makes extensive use of, a class called ImmutableList. Figure 1.4 shows a small example of using this immutable data structure. The key concept is that when a new object is 'added' to an immutable data structure, a new data structure is created that contains both the original contents and the new object. The method that does this here is called append.

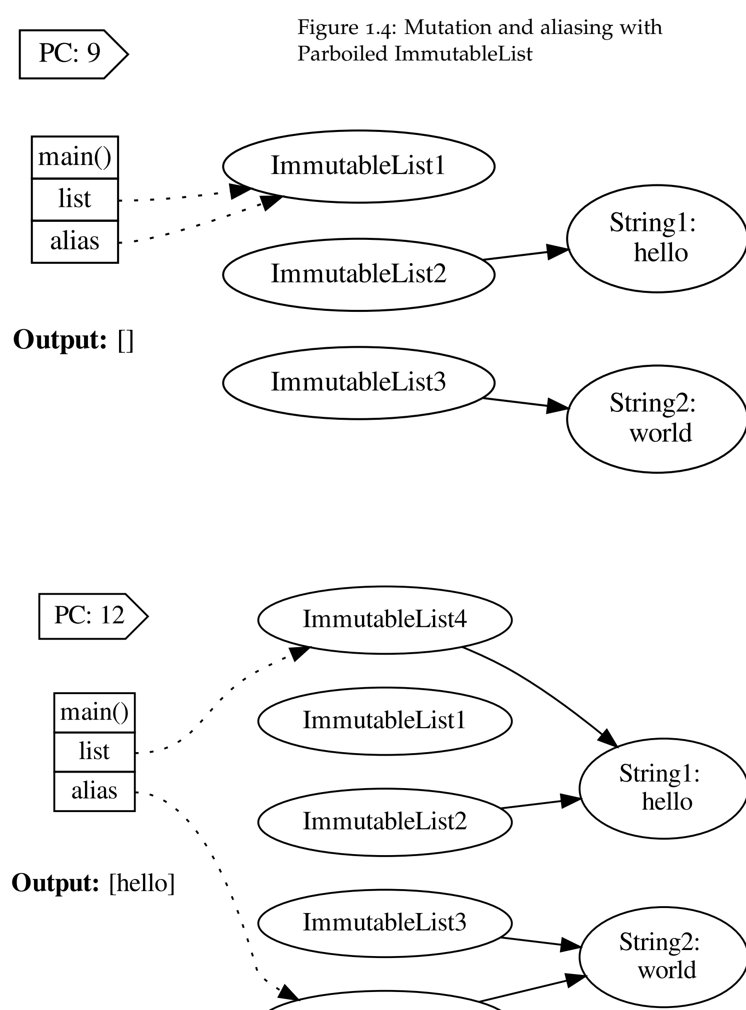
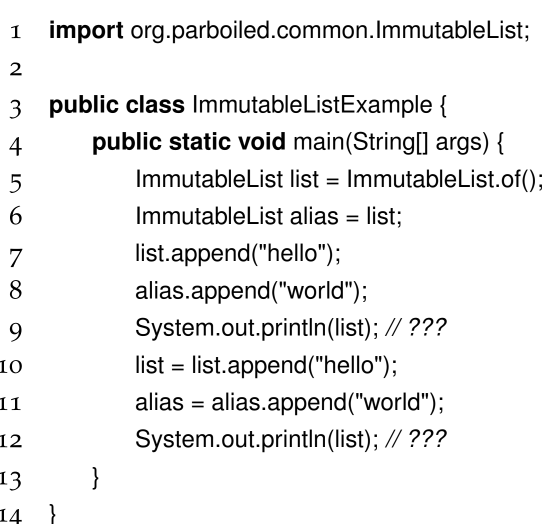


Figure 1.4: Mutation and aliasing with Parboiled ImmutableList



```
1  import org.parboiled.common.ImmutableList;
2
3  public class ImmutableListExample {
4      public static void main(String[] args) {
5          ImmutableList list = ImmutableList.of();
6          ImmutableList alias = list;
7          list.append("hello");
8          alias.append("world");
9          System.out.println(list); // ???
10         list = list.append("hello");
11         alias = alias.append("world");
12         System.out.println(list); // ???
13     }
14 }
```

1.1.7 *The keyword 'final' prevents re-assignment*

In Java the keyword final prevents re-assignment. When the programmer uses 'final', the compiler is obliged to prove that the variable is assigned exactly once on every program path. Figure 1.5 illustrates four different correct usages of the final keyword, whereas Figure 1.6 illustrates an incorrect usage. Combining immutable data structures with the final keyword can make it easier to reason about the program, because we will know that the values were set in exactly one place and didn't change after that.

The Java keyword final is similar to, but not exactly the same as, the C/C++ keyword const.

We will learn how the compiler does this proof — using dataflow analysis — after the midterm.

```
1  public class Final {
2      // procedure parameters can also be declared final
3      public static void main(final String[] args) {
4          final String s = "known";
5          // the javac compiler will prove that msg is
6          // assigned exactly once on every program path
7          final String msg;
8          if (args.length == 0) { msg = "no args"; }
9          else { msg = "some args"; }
10         final C c = new C(msg);
11         System.out.println(c.f);
12     }
13 }
14 class C {
15     final String f; // compiler will prove that f is assigned exactly once in every constructor
16     C(final String p) { this.f = p; }
17 }
```


Figure 1.5: Correct usage of the 'final' keyword

```
1 public class Bogus {
2     public static void main(String[] args) {
3         final String s = "setting it now";
4         s = "trying to re−assign it −−− won't work!";
5     }
6 }
```

```
1 $ javac Bogus.java
2 Bogus.java:4: error: cannot assign a value to final variable s
3         s = "trying to re−assign it −−− won't work!";
4         ^
5 1 error
```

Figure 1.6: Incorrect usage of the 'final' keyword

1.2 Algorithmic Complexity

Big-O notation is used to express the asymptotic complexity of an algorithm, usually in terms of time, but sometimes in terms of space. Big-O notation is sometimes referred to as *computational complexity* or *asymptotic complexity*. Let n be a measurement of the size of the input.

Big-O	*Name*		*Example*
$O(1)$	constant	(polynomial)	hash-table lookup
$O(\lg n)$	logarithmic	(polynomial)	binary search
$O(n)$	linear	(polynomial)	bucket sort
$O(n \lg n)$	log-linear	(polynomial)	quick sort
$O(n^2)$	quadratic	(polynomial)	bubble sort
$O(n^3)$	cubic	(polynomial)	Early's parsing algorithm
$O(2^n)$	NP-complete	(exponential)	sat
—	undecidable		the halting problem

Figure 1.7: Summary of some big-O related terminology. Anything less than exponential we refer to as *polynomial*. In ece250 you learned a variety of polynomial time algorithms, mostly for sorting and searching. In that context, the difference between $O(n^2)$ (*e.g.*, bubble sort) and $O(n \lg n)$ (*e.g.*, quick sort) is important. In ece351 we are primarily concerned with the difference between polynomial and exponential (and undecidable). From the perspective of ece351, all polynomial time problems are easy.

The dominant term is all that matters in big-O notation. For example, $O(n^2 + n)$ is just written as $O(n^2)$ because the n^2 term dominates the n term.

1.2.1 NP-Completeness and Boolean sat*isfiability*

NP stands for *non-deterministic polynomial time*:

a. If we have a solution, then we can verify it in polynomial time on a single-core deterministic machine.
b. If we have a magical (*i.e.*, non-determinisitic) single-core machine that can always guess the right next step (*i.e.*, has 'non-deterministic powers'), then we can compute a solution in a polynomial number of magically correct guesses.
c. If we have a massively parallel deterministic computer that has an exponential number of cores, then we can compute a solution in polynomial time: every core tries a different path, and some of them will find a solution (if a solution exists) in a polynomial number of steps.
d. If we have a single-core deterministic computer, then we can compute a solution in exponential time.

The canonical example of an NP-complete problem is the Boolean satisfiability problem, commonly known as sat. Given a boolean formula, find values for the variables that make the formula true. The brute force way to solve this problem is to enumerate the truth table for the function and look for a row that evaluates to true. The

A *boolean formula* here is essentially an $\mathcal{F}$ program: the variables have only boolean (true/false) values, and the only operators are conjunction (logical and), disjunction (logical or), and negation (logical not).

truth table is exponential in the number of variables in the formula: *e.g.*, a formula with three variables will have 2^3 rows in its truth table. Consider, for example, the boolean formula $x \cdot y \cdot z$ depicted in Figure 1.8.

x	$\cdot y$	$\cdot z$	$= f$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

Figure 1.8: Truth table for $x \cdot y \cdot z$. The formula has 3 variables and the table has 2^3 rows, only one of which evaluates to true (the shaded row at the bottom of the table).

A non-deterministic machine could 'magically' (non-deterministically) choose the right row in a single step. You can think of non-deterministic powers as parallelism here. If a machine had eight cores it could evaluate all eight rows of the truth table in parallel and thereby determine which rows evaluated to true in a small amount of clock time.

A single core (*i.e.*, deterministic) machine would have to evaluate each row of the table in sequence, which in the worst case could take an exponential amount of time (because there are an exponential number of rows in the table).

A defining feature of NP-complete problems is not just that they take an exponential amount of time on a deterministic machine, but that the answer to such a problem can be verified in polynomial time. For example, if someone tells us that $\langle 1, 1, 1 \rangle$ is a solution to the formula $x \cdot y \cdot z$, we can easily verify that by substituting the values into the formula.

While SAT is the canonical example of an NP-complete problem, there are other common examples, including: travelling salesman, scheduling, knapsack/bin packing, and graph colouring. All of these kinds of problems can be used to encode SAT (which is the standard technique for proving that a problem is NP-complete). When faced with a new problem of unknown complexity you want to think if it is similar to any of these known NP-complete problems. For example, register allocation appears similar to bin packing.

In this course we will see a number of NP-complete problems: register allocation by graph colouring; circuit equivalence by graph isomorphism, *etc.*

1.2.2 *Undecidability and The Halting Problem*

There are limits to what can be computed. Problems that cannot be solved on a computer are known as *undecidable*.

The canonical example of an undecidable problem is *The Halting Problem:* given a program P and an input for that program i, prove that P will not get stuck in a loop when it executes with i. In other words, prove that program P will *halt* on input i.

Prof Craig Kaplan in CS has a nice web page about the Halting Problem with code examples at `http://www.cgl.uwaterloo.ca/~csk/halt/`
The Wikipedia page is also good `http://en.wikipedia.org/wiki/Halting_problem`

In the 1930's Alan Turing famously devised a program P and input i for that program where he proved that it was impossible to prove that P would halt on i. Because there exists one case it which

this problem is undecidable we say that it is undecidable in general. There are lots of programs for which it easy to prove termination (*i.e.*, that they will halt on all possible inputs). For example, the classic hello world program, or any other program without loops and without recursion, will always halt for any input.

An important corollary to the Halting Problem proof is that given two programs P and Q (written in some Turing Complete language), we cannot decide, in general, if P and Q are equivalent. This has important implications for compilers. How can we test our compilers if we cannot compute equivalence of their outputs? Suppose you add a new optimization to your compiler: how can you know that it is correct if you can't compute the equivalence of the optimized program to the original program?

Turing Completeness is defined in §2.1.1.

In practice what compiler engineers do is check that the compiled programs compute the same outputs. Let's name our two source programs P and Q. Let P' and Q' name the compiled versions of P and Q (*i.e.*, the output of our compiler). Then we run P' on input i to get output p, and we run Q' on input i to get output q, and finally we check that p and q are equivalent. This is not as good as directly checking P and Q for equivalence, because there might be some input x that we did not test for which P' and Q' compute different outputs.

1.2.3 Easy, Hard, and Impossible

For the purposes of this course, polynomial time algorithms are *easy*, exponential time algorithms are *hard*, and undecidable problems are *impossible*. Note that this classification is about the amount of time that the computer must spend running the algorithm, not about how much time and cognition the engineer must expend implementing the algorithm. For example, quick sort is a tricky algorithm to implement, but it runs quickly, so here we would say that it is 'easy'.

There are a number of hard (NP-complete) problems in compiler engineering. There are at least three ways to solve such problems:

a. Use a polynomial time approximation algorithm. This is the approach most commonly used in practice. Any compiler textbook will describe a number of polynomial approximations for a hard problem.

b. Implement an NP-complete algorithm yourself. This is never done in compilers, and almost never done anywhere else.

c. Translate the problem into a boolean formula and ask a SAT solver for the answer. This is the direction that research is heading in, and it is what we recommend that you do in your future career

In this course we will not study any polynomial time approximation algorithms, because we will not implement any solutions to hard problems in the labs. Humans can solve small NP-complete problems on paper, which is what we will do on the exam. Once you understand the hard problem you can always go and learn some polynomial approximation for it on your own later.

if faced with an NP-complete problem for which there is not a well-accepted polynomial time approximation algorithm.

We compute equivalence of $\mathcal{F}$ programs by translating them to SAT and asking a SAT-solver for the answer.

There has been several decades of intense research on SAT-solvers, including annual competitions with various prize categories. SAT-solvers are used in a variety of areas in industry, but especially in digital hardware design tools. It is highly unlikely that a future you, working under time and budget constraints, will be able to write a solver for an NP-complete problem that runs more quickly than one of the existing open-source SAT-solvers.

1.3 *Four Variants of Linear Search*

Figure 1.9 lists four different implementations of linear search. The input, output, and algorithm are the same for all:

- *Input:* a list of strings, where the first one is the token to search for in the rest of the list.
- *Output:*
  - Normal: prints "found it!" or "didn't find it"
  - Errors: if the list is not provided
- *Algorithm:* linear search
  - Runtime: *O(n)*
  - Space consumption: *O(n)*
  - Termination: either the token is found or the end of the list is reached.

So what's differs between these four programs?

- *Static Structure:*
  - Data Structures: arrays *vs.* linked lists
- *Dynamic Execution:*
  - Control Structures / Call Graph: iteration *vs.* recursion

EXERCISE: draw an object diagram of the state of each program when it finds a match given the input 'Moe Larry Curly Moe'. Solutions given on subsequent pages in Figures 1.10, 1.11, 1.12, 1.13.

```
1  public class LinearSearch_Array_Iterative {
2      public static void main(String[] args) {
3          String toFind = args[0];
4          for (int i = 1; i < args.length; i++) {
5              if (toFind.equals(args[i])) {
6                  System.out.println("found it!");
7                  System.exit(0);
8              }
9          }
10         System.out.println("didn't find it");
11         System.exit(1);
12     }
13 }
```

```
1  public class LinearSearch_Array_Recursive {
2      public static void main(String[] args) {
3          String toFind = args[0];
4          search(toFind, args, 1);
5          System.out.println("didn't find it");
6          System.exit(1);
7      }
8      static void search(String toFind, String[] a, int i) {
9          if (toFind.equals(a[i])) {
10             System.out.println("found it!");
11             System.exit(0);
12         } else {
13             if (i < a.length−1) search(toFind, a, i+1);
14         }
15     }
16 }
```

```
1  public class LinearSearch_Objects_Iterative {
2      public static void main(String[] args) {
3          String toFind = args[0];
4          // copy array to linked list
5          Node head = null;
6          for (int i = args.length−1; i > 0; i−−) {
7              head = new Node(head, args[i]);
8          }
9          // now search
10         Node n = head;
11         while (n != null) {
12             if (toFind.equals(n.data)) {
13                 System.out.println("found it!");
14                 System.exit(0);
15             } else {
16                 n = n.next;
17             }
18         }
19         // search is over: unsuccessfull
20         System.out.println("didn't find it");
21         System.exit(1);
22     }
23 }
```

```
1  public class LinearSearch_Objects_Recursive {
2      public static void main(String[] args) {
3          String toFind = args[0];
4          // copy array to linked list
5          Node head = null;
6          for (int i = args.length−1; i > 0; i−−) {
7              head = new Node(head, args[i]);
8          }
9          // now search
10         search(toFind, head);
11         // search is over: unsuccessfull
12         System.out.println("didn't find it");
13         System.exit(1);
14     }
15     static void search(String toFind, Node n) {
16         if (n == null) { return; }
17         else {
18             if (toFind.equals(n.data)) {
19                 System.out.println("found it!");
20                 System.exit(0);
21             } else {
22                 search(toFind, n.next);
23             }
24         }
25     }
26 }
```

Figure 1.9: Four different implementations of linear search.

```
1  public class LinearSearch_Array_Iterative {
2      public static void main(String[] args) {
3          String toFind = args[0];
4          for (int i = 1; i < args.length; i++) {
5              if (toFind.equals(args[i])) {
6                  System.out.println("found it!");
7                  System.exit(0);
8              }
9          }
10         System.out.println("didn’t find it");
11         System.exit(1);
12     }
13 }
```

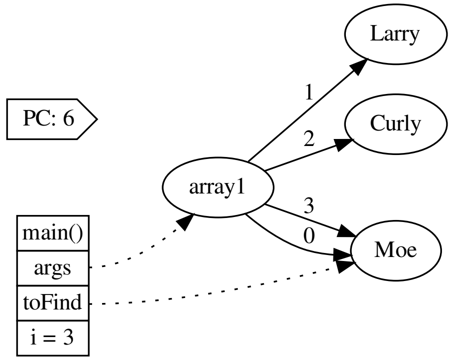


Figure 1.10: Object diagram for iterative array variant of linear search when it finds ‘Moe’ in ‘Larry Curly Moe’.

```
1  public class LinearSearch_Array_Recursive {
2      public static void main(String[] args) {
3          String toFind = args[0];
4          search(toFind, args, 1);
5          System.out.println("didn't find it");
6          System.exit(1);
7      }
8      static void search(String toFind, String[] a, int i) {
9          if (toFind.equals(a[i])) {
10             System.out.println("found it!");
11             System.exit(0);
12         } else {
13             if (i < a.length−1) search(toFind, a, i+1);
14         }
15     }
16 }
```

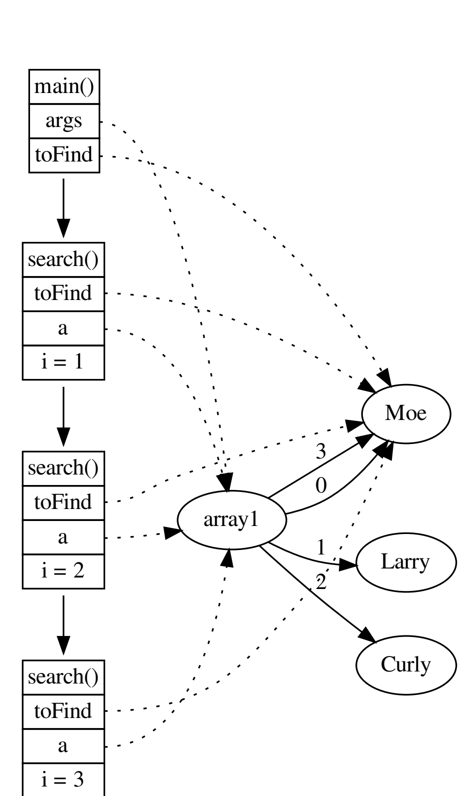


Figure 1.11: Object diagram for recursive array variant of linear search when it finds 'Moe' in 'Larry Curly Moe'.

```
1  public class LinearSearch_Objects_Iterative {
2      public static void main(String[] args) {
3          String toFind = args[0];
4          // copy array to linked list
5          Node head = null;
6          for (int i = args.length−1; i > 0; i−−) {
7              head = new Node(head, args[i]);
8          }
9          // now search
10         Node n = head;
11         while (n != null) {
12             if (toFind.equals(n.data)) {
13                 System.out.println("found it!");
14                 System.exit(0);
15             } else {
16                 n = n.next;
17             }
18         }
19         // search is over: unsuccessfull
20         System.out.println("didn't find it");
21         System.exit(1);
22     }
23 }
```

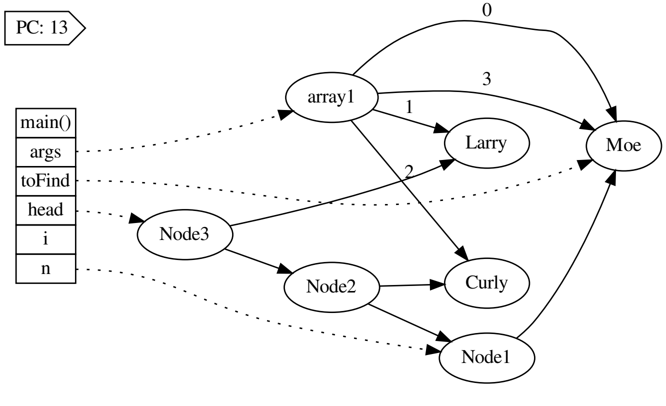


Figure 1.12: Object diagram for iterative linked-list variant of linear search when it finds 'Moe' in 'Larry Curly Moe'.

```
1  public class LinearSearch_Objects_Recursive {
2      public static void main(String[] args) {
3          String toFind = args[0];
4          // copy array to linked list
5          Node head = null;
6          for (int i = args.length−1; i > 0; i−−) {
7              head = new Node(head, args[i]);
8          }
9          // now search
10         search(toFind, head);
11         // search is over: unsuccessfull
12         System.out.println("didn’t find it");
13         System.exit(1);
14     }
15     static void search(String toFind, Node n) {
16         if (n == null) { return; }
17         else {
18             if (toFind.equals(n.data)) {
19                 System.out.println("found it!");
20                 System.exit(0);
21             } else {
22                 search(toFind, n.next);
23             }
24         }
25     }
26 }
```

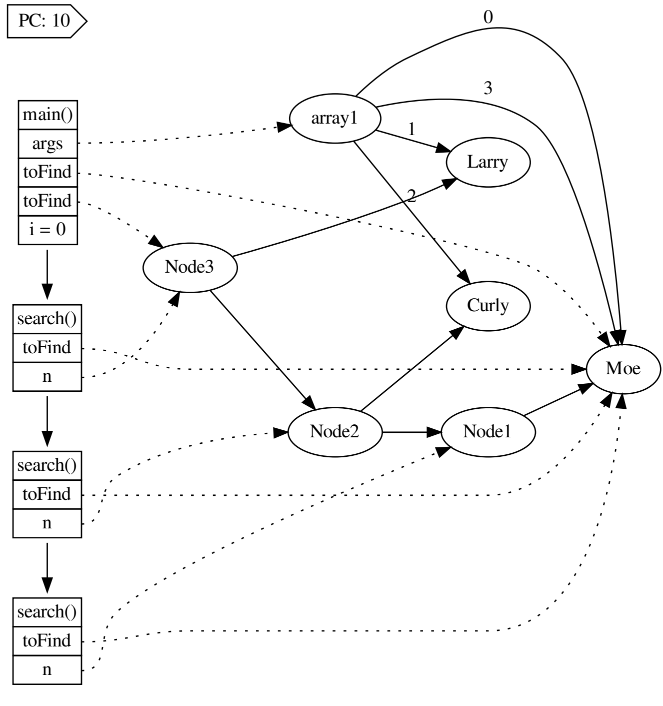


Figure 1.13: Object diagram for recursive linked-list variant of linear search when it finds ‘Moe’ in ‘Larry Curly Moe’.

1.4 Programming 'Paradigms'

There are different styles of programming languages. These have historically been referred to as programming *paradigms* because languages tended to fit neatly into one or the other category. Many modern languages, however, have features from a variety of styles. One of the major trends of the last decade is that mainstream object-oriented languages are acquiring features historically associated with functional languages.

In this course we will write in Java, an imperative object-oriented language with some functional features. You will learn more advanced object-oriented programming techniques (such as *design patterns*) and functional programming techniques (such as *immutable data*) than you have been previously exposed to.

The term *paradigm shift* was coined by historian of science Thomas Kuhn in his book *The Structure of Scientific Revolutions*. His hypothesis was that, for example, Newton's account of planetary motion was so different from Ptolemy's account that the two scientists wouldn't even be able to talk to each other to compare notes, because they were mentally in two different paradigms.

The term 'programming paradigm' was coined at a time when it seemed (to some people at least) that Lisp/Scheme programmers and C programmers would never be able to understand each other, and there would be no way to combine ideas from the two languages.

The view today is best summed up by Erik Meijer, architect of Microsoft's LINQ technology, in his paper *Confessions of a used programming language salesman*: 'functional programming has finally reached the masses, except that it is called Visual Basic 9 instead of Haskell 98.'

Imperative languages have two distinctive features:

a. *Sequential composition.* Statements are executed one after the other. Other kinds of languages might not even have statements, and the evaluation order of the expressions might be determined by the compiler, rather than being specified by the programmer as in an imperative languages.

b. *Re-assignment.* The value of variables can be changed. Here is an example program exhibiting both sequential composition and re-assignment:

```
x = 7; // initial assignment to x
print(x); // print 7 to console
x = 2; // re−assignment: change the value of x to 2
print(x); // print 2 to console
```

The theoretical model for imperative languages is Turing Machines.

Functional languages are characterized by three things:

a. *Immutable Data*. Variables cannot be re-assigned.

b. *Anonymous Functions*. Often denoted with lambda (λ). Java 8 and C++ 11 have both added anonymous functions. For example, here is an anonymous function that takes two arguments (x and y) and returns 'true' if x is less than y:

```
λ x, y . x < y
```

c. *Higher-Order Functions*. A higher-order function is a function that takes a function pointer as an argument. A common example is the sort function, which often takes two arguments: the list to be sorted and a comparator function (that defines if the sort should be ascending or descending). In a language with anonymous functions we could call sort and specify ascending order with an anonymous function like so:

```
comparator = λ x, y . x < y;
sort(list, comparator);
```

The theoretical model for functional languages is the λ calculus. This is theoretically equivalent to Turing Machines, but emphasizes a style based on expressions and immutable values (like math), rather than on manipulating the state of a machine (imperative languages).

There is sometimes some confusion about the intended meaning of the term *functional*. The intended meaning is from mathematics: a mathematical function computes a single output based solely on the values of its (immutable) input arguments. In English, the word *functional* also means *useful*, and while that connotation is intended it is not what defines functional programming languages in contrast to other kinds of languages.

You are welcome to use lambda expressions in your code. We have not included any lambdas in the skeleton code because some students are not yet familiar with it, and also because it is difficult to trace in the debugger.

Paradigm	*Characteristics*	*Example Languages*
Imperative	has assignment has sequential composition	Pascal, Basic, Turing, C, Java, C++, C#
Object-oriented	has objects might have classes might have inheritance might have dynamic dispatch	Java, C++, C#, Scala, Python, Javascript, Smalltalk, OCaml, Ruby
Functional	has higher-order functions has anonymous functions (lambda λ) data are immutable by default	Lisp/Scheme, Haskell, OCaml, F#, Python, Scala
Logic	has logical equations/expressions engine to evaluate equations (*e.g.*, iteration to a fixed point)	Prolog, SQL

Figure 1.14: Programming paradigms

Pragmatics: §2.2.1 + §2.4.1
Tiger: §2.3–5

Chapter 2
Regular Languages & Finite Automata

Regular languages are important
- *theory* — simplest kind of formal language
- *practice*
  - *easy* — implemented with finite automata
  - *useful* — for many tasks

Finite automata are the simplest kinds of machines: they have no storage capability beyond their current state. More sophisticated machines have additional storage, such as a stack (pushdown automata) or a tape (Turing Machine, Linear Bounded Automata).

More sophisticated machines, that we will study later, have richer storage.

Regex	Regular expression
Automata	Machine
DFA	Deterministic Finite Automaton
NFA	Nondeterministic Finite Automaton

Figure 2.1: Terminology for FSA/FSM/*etc.*

2.1 *Kinds of Machines and Computational Power*

The *computational power* of a machine is the set of problems that it can compute. We say machine X is more powerful than machine Y if there are some problems that X can solve that Y cannot solve.

There are four kinds of theoretical machines that are of interest to us. Their computational power grows along with the amount of storage available to them, and the flexibility in accessing that storage. In all cases the machine's controller has a finite number of states.

Finite State Machines (FSM): You should be familiar with these from other courses (*e.g.*, ECE327). FSMs always run in linear time, because at each step they consume one token of the input. FSMs are what we use to recognize regular languages.

Deterministic FSMs (DFAs) are equivalent in power and speed to nondeterministic FSMs (NFAs): both run in linear time. An NFA might have exponentially fewer states than a DFA.

Push-down automata (PDA): A PDA is a FSM plus an (infinite) stack for storing values. Push-down automata are the kind of machines needed to recognize context-free languages. Push-down automata always run in polynomial time.

Non-deterministic PDAs are more powerful than deterministic PDAs, although we will not exploit that fact in this course. Some context-free grammars cannot be parsed by a deterministic PDA and require a non-deterministic PDA.

Linear-bounded automata (LBA): A LBA is a FSM plus a finite tape. The physical computers we use every day are, from a strict theoretical perspective, LBAS, even though we often think of them as Turing Machines.

Turing Machine TM: A Turing Machine is a FSM plus an (infinite) tape. We usually conceptualize the computers we use every day as Turing machines, even though technically they are LBAS.

Non-determinism gives a Turing Machine speed but not power. The set of problems that can be computed is the same, but the non-deterministic machine can compute some of them exponentially faster. These are the NP-complete problems: *non-deterministic polynomial time*, or exponential time on a deterministic machine.

The idea of a Turing Machines is a thought experiment that Alan Turing came up with in the 1930s. These are not machines that existed as such. At that time a 'computer' was a person who worked with a slide rule, pencil, and paper.

2.1.1 Turing Machines and Programming Languages

We say a programming language is *Turing complete* if it can be used to describe any computation that can be done by a Turing Machine. Roughly speaking, if the programming language has conditionals (if statements), loops or recursion, and a potentially unlimited number of variables then it is Turing complete.

Almost everything that you think of as a 'programming language' is Turing complete. Languages like HTML and SQL are not Turing complete, and for that reason are sometimes considered to not be programming languages.

The *Turing tar-pit* refers either to programming languages that lack higher level constructs, or to the argument that such constructs are unnecessary because the programming language under consideration is already Turing complete. Higher-level constructs do not add to the theoretical expressive power of a language that is already Turing complete, but they might make a person writing in that language more productive.

54. Beware of the Turing tar-pit in which everything is possible but nothing of interest is easy. — Alan Perlis, Epigrams on Programming, 1982

2.2 Regular Expressions

Regular expressions are the simplest class of formal languages, and define what can be computed by a finite automaton. A regular expression is defined over some *alphabet*, named Σ. There are three operations that can be used to compose smaller regular expressions into larger ones:

You might be familiar with *regular expressions* from co-op work.

In practice, there are some extensions that can be defined in terms of these three base operations. For example, $+$ as one or more can be defined with concatenation and star: $A^+ = AA^*$.

Concatenation	One thing followed by another: *e.g.*, AB.
Alternation	One of two alternatives: *e.g.* $A|B$ means A or B.
Repetition	Zero or more: A^*

The expressive power of regular languages is exactly the same as what can be computed with finite automata.

Grep is a Compiler. From a user's perspective, grep is a program that searches for a string in a text file. Really, it's a compiler in disguise. A compiler transforms a source program into an executable program. Grep transforms a regular expression (its source program) into a finite state machine (which can be executed). In this chapter we will learn the basic theory behind how grep (and similar tools) do this compilation.

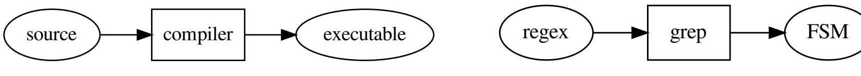


Figure 2.2: A compiler transforms a source program to an executable program. Grep transforms a regex to a finite state machine.

Grep is an Interpreter. Does grep actually return the executable finite state machine? No. What it returns is the set of lines in the input text file that match the regular expression. So it is more accurate to say that grep is an interpreter.

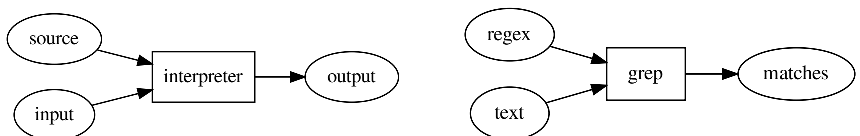


Figure 2.3: An interpreter evaluates (*i.e.*, interprets) a source program on a specific input and returns the output computed by the source program.

Grep is a Just-In-Time Compiler. An interpreter evaluates the source program directly: it does not translate it into an executable form. Grep, internally, produces an executable finite state machine (fsm). So grep is really more like a *just-in-time* (jit) compiler: it produces executable code from an input program, runs that code on a particular input, returns the output and discards the generated code. This is what modern virtual machines, such as Oracle's jvm or Microsoft's clr, do. So it is most accurate to say that grep is a just-in-time compiler.

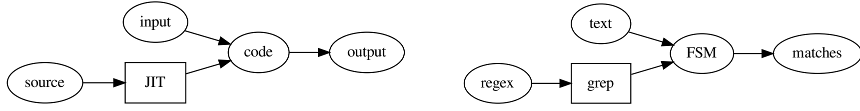


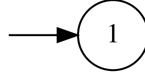
Figure 2.4: An interpreter evaluates (*i.e.*, interprets) a source program on a specific input and returns the output computed by the source program.

2.3 Finite State Machines

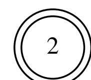
A *finite state machine* (*i.e.*, finite automata) reads characters from an input tape. When it reaches the end of the input tape, it returns one of two results: either it *accepts* or *rejects* the string on the input tape. The string on the input tape will contain characters from the machine's *alphabet*.

Example input tape:

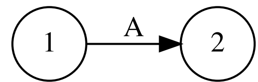
Finite state machines are commonly visualized with circles for *states* and labelled edges for *transitions* between states.



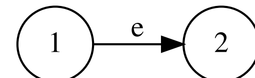
Initial State. A state with an incoming arrow that has no origin. There will be just one initial state for each machine.



Accepting State. A double circle. There might be multiple accepting states. Note that execution of the machine does not terminate when an accepting state is reached: it terminates when the end of the input is reached. If the machine happens to be in an accepting state at that time, then the machine accepts that input.



Transition. An arrow between two states, labelled with the input character that causes the machine to take that transtion. In this example, if the machine is in state 1 and the next character in the input is an 'A', then it will transition to state 2.



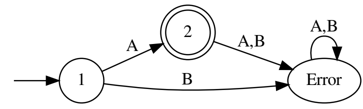
Epsilon Transition. *Non-deterministic* finite state machines include *epsilon (ϵ) transitions*. This is a transition that the machine can take without consuming an input character. There might be multiple outgoing epsilon transitions from a single state. It is understood that the machine will 'magically' choose the epsilon transition that will eventually lead to accepting the input — if it is possible for the machine to accept the given input.

In our diagrams we will sometimes label epsilon transitions with a lower-case *e* rather than with an ϵ.

Crafting uses lambda (λ) instead of epsilon (ϵ). Most books use ϵ.

Deterministic finite automata do not have epsilon transitions.

Implicit Error State. There is, implicitly, a distinguished error state that the machine transitions to when the next character of the input does not match any of the available transitions. Consider, for example, a machine that accepts the string 'A', from the alphabet {A, B}. What happens if this machine receives the input string 'B'? It rejects, of course — it goes to the Error state. We can explicitly draw this Error state, and then add transitions to it from every other state in the machine, labelled with every character of the alphabet that they do not already have a transition for. This tends to make the diagrams very cluttered, so typically the Error state is left implicit (*i.e.*, not visualized).



Finite state machines are the simplest kind of theoretical machine. They have no storage beyond their current state. More sophisticated kinds of machines, such as push-down automata and Turing Machines, add more storage to the foundational finite automata model.

2.4 *Regex* → NFA

This material is covered well in all compiler text books. The conversion rules from the Tiger book and the PLP books are reproduced here for your convenience in Figure 2.5 and Figure 2.6. *Crafting* has equivalent figures with slightly different visual notation.

Pragmatics: p.56

Tiger: p.25

Crafting: §3.8.1. Note that *Crafting* uses λ for the empty string, whereas the other two books use ϵ.

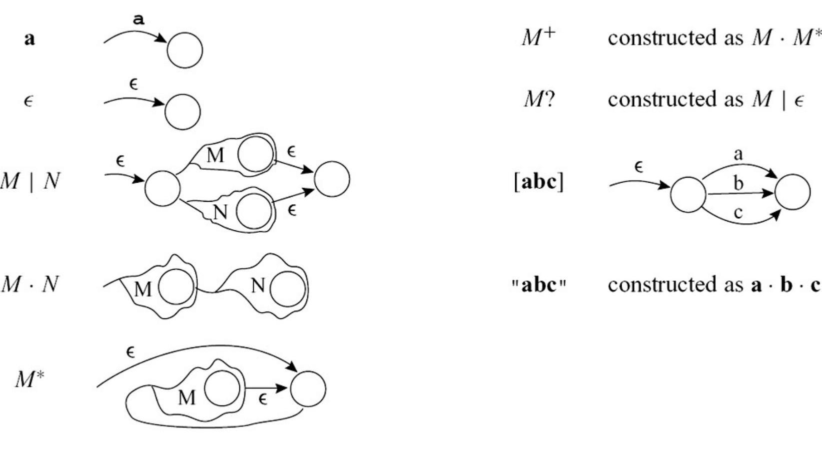
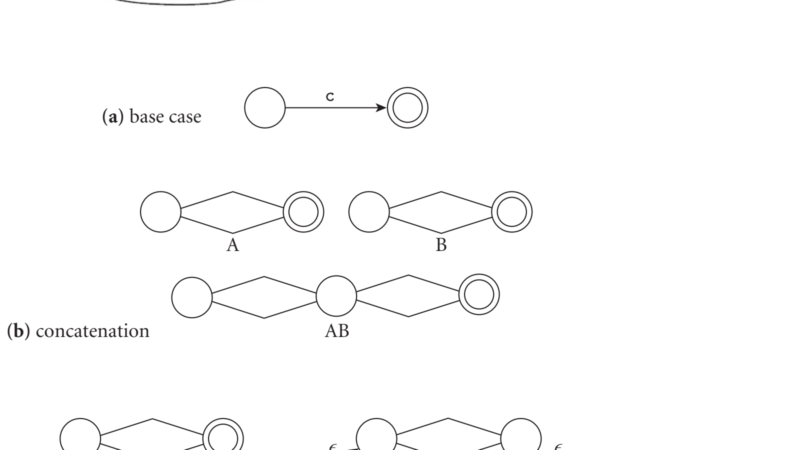


Figure 2.5: Rules for converting regex to NFA [Tiger f2.6]



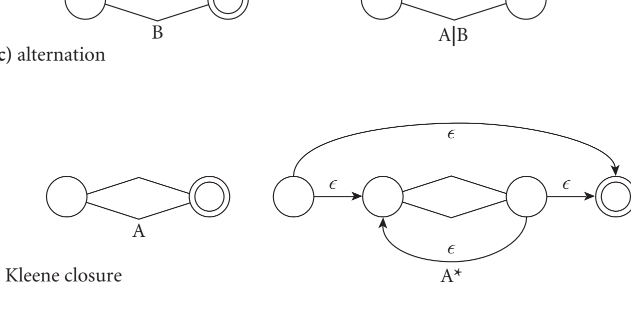


Figure 2.6: Rules for converting regex to NFA [PLP f2.7]

2.5 NFA → DFA

Pragmatics: p.57–58
Tiger: p.27
Crafting: §3.8.2

All three textbooks use the standard 'set of subsets' approach. *Crafting* describes the approach in words and pseudo-code, whereas the other two books just use words.

In the worst case, the output DFA might be exponentially larger than the input NFA. This case rarely happens in practice.

a. Create initial DFA state A, which comprises the NFA start state plus all NFA states that are reachable from its initial state by ϵ transitions.

ϵ^*

b. For each letter x in the NFA's alphabet, see which DFA states it transitions to from the DFA states included in A. Which states can those states get to via ϵ? Note that the originating NFA states to be considered here are just the ones that consume x (for each x).

$x\epsilon^*$ (not $\epsilon^* x$)

2.6 DFA *Minimization*

Pragmatics: p.59
Crafting: §3.8.3
Tiger: not covered.

The DFA produced in the above procedure is often not as small as it could be. We can use the following procedure to attempt to minimize it. In following this procudure, we will, during intermediate steps, construct machines that are not legal DFA's. However, at the end of the procudure we will have a legal DFA, and one that is hopefully smaller than what we started with.

a. Merge all final states into a new final state. Merge all non-final states into a new non-final state. This machine is likely not a legal DFA, because it will have states with multiple transitions with the same label; call these *ambiguous* transitions.

b. Pick an ambiguous transition. Split its start state so that transition is no longer ambiguous. Repeat until all ambiguities have been removed.

In the example from Figure 2.8b:

a. Merge all non-final states: ABC. Merge all final states: $DEFG$.

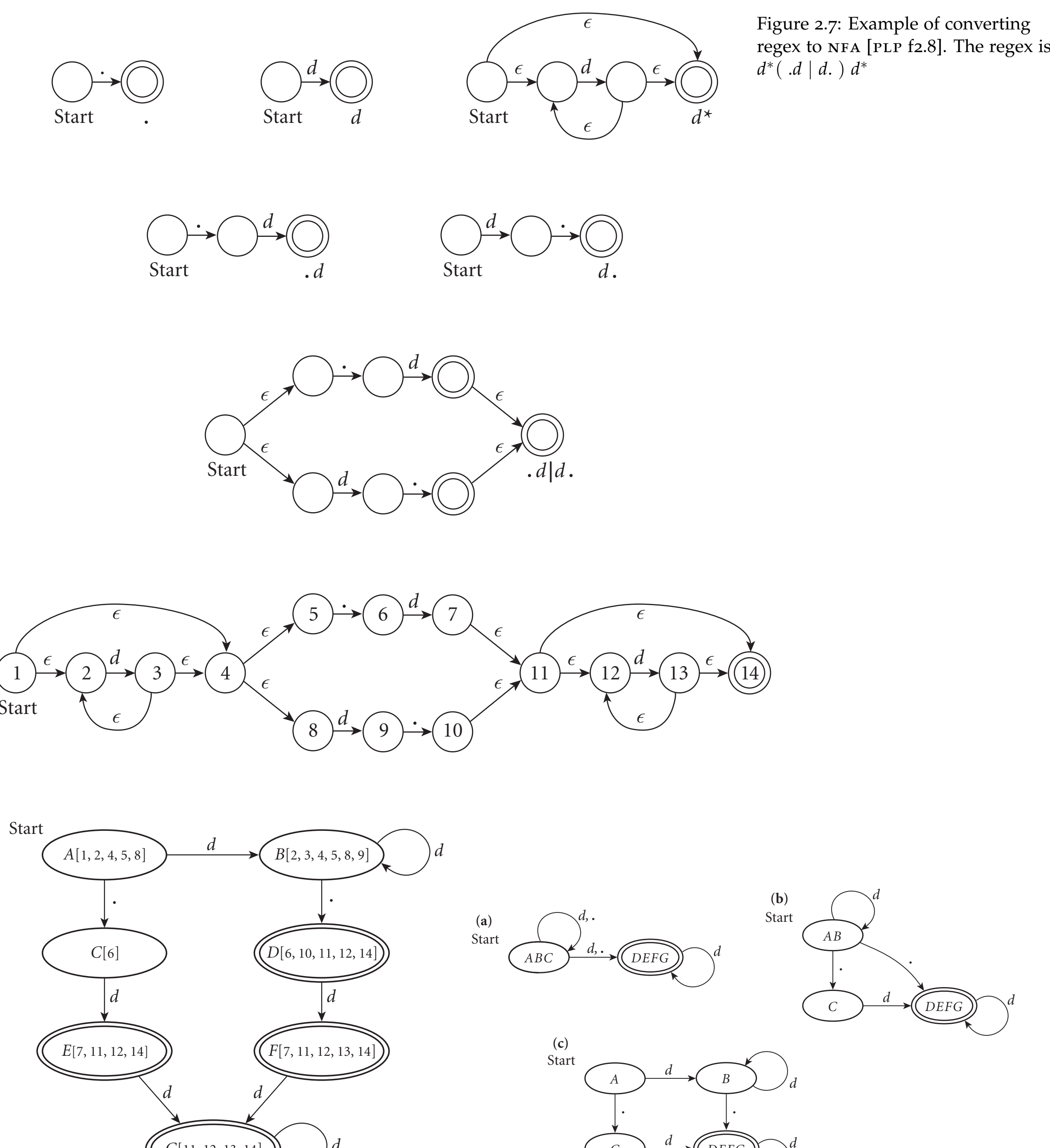
b. ABC has an ambiguous transition on 'd': split it into AB and C to remove the ambiguity. We know that A and C must be split from each other by looking at the DFA: A and C both accept d but have different targets (B and D, respectively). Must we also split B to its own state? No, we can group it with A because they share the same target when accepting a d, namely B.

c. AB has an ambiguous transition on '.': split it into A and B to remove the ambiguity. No ambiguities remain. Done.

2.7 *Example regex* → NFA → DFA → *minimized* DFA

Pragmatics: p.59

Figure 2.7: Example of converting regex to NFA [PLP f2.8]. The regex is: $d^*(.d \mid d.)\ d^*$



(a) NFA → DFA
Error: NFA state 11 should not be in DFA state G. `http://www.cs.rochester.edu/u/scott/pragmatics/3e/errata.shtml`

(b) DFA minimization

Figure 2.8: Example of NFA to DFA conversion and DFA minimization [PLP f2.9 & f2.10]

2.8 DFA *Minimization with an Explicit Error State*

The above presentation of DFA minimization works most of the time, but not always. There is one more pre-step that is sometimes needed: adding a distinguished error state to the DFA before minimization.

Consider the regular expression `f?g*`, for zero or one fs followed by any number of gs. We could derive the following NFA, where e edges are ϵ edges, and the corresponding non-minimal DFA:

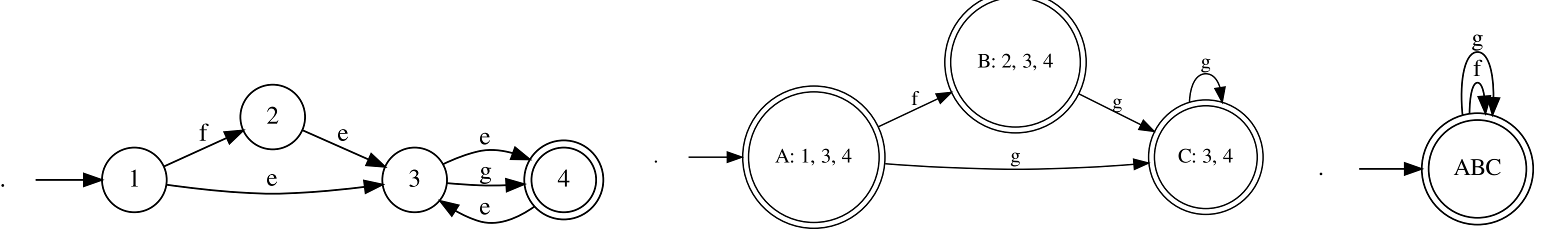


Figure 2.9: An NFA and corresponding non-minimal DFA for the regular expression `f?g*`, as well as a bogus minimized DFA.

This is not good, this is not right: this 'minimized' DFA is not equivalent to the original language — it accepts any number of fs and and number of gs in any order. What gives? We need to explicitly represent error transitions in the non-minimal DFA before minimizing.

What happens to the non-minimal DFA if it gets an f while in state B? It rejects. We can model this explicitly by introducing a terminal non-accepting state X, like so:

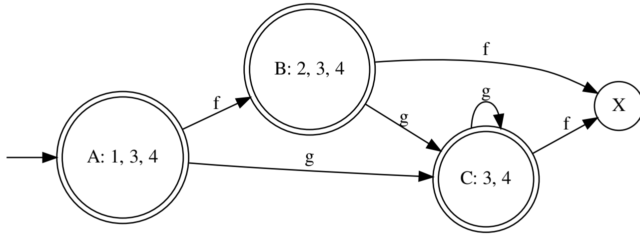
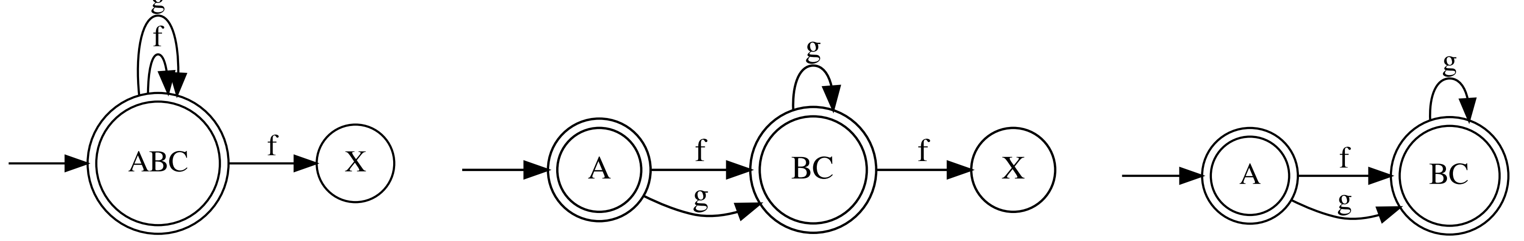


Figure 2.10: DFA with an explicit error state X.

Now we have explicitly specified what every state does for each letter of the input alphabet (*i.e.*, f and g). If we apply our minimization procedure to this error-explicit DFA we will get the right answer:



(a) merge final and non-final states

(b) split A from BC to remove ambiguity on f

(c) drop explicit error state X

Figure 2.11: Minimizing a DFA with an explicit error state.

2.9 *Another Example*

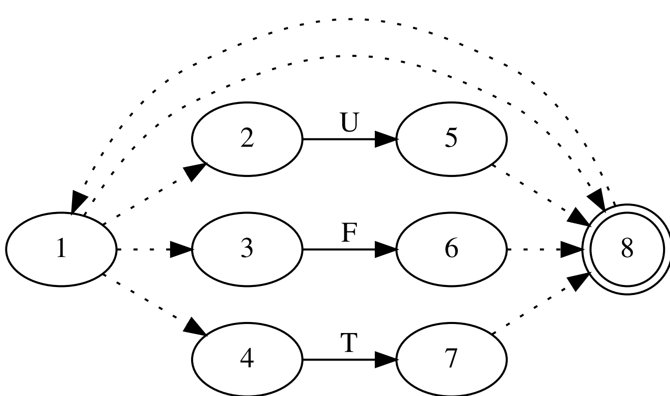
Recall the $\mathcal{W}$ language for Boolean waveforms from the labs. Imagine that we extend this language for a three-valued logic: true, false, and unknown. Here is a regular expression for this new language. Complete each construction and transformation.

From a previous midterm.

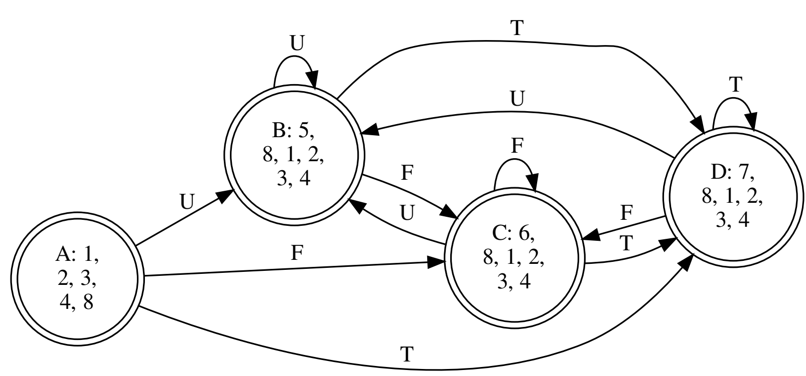
(U|F|T)*

2.9.1 *Regex → NFA*

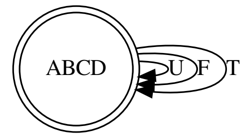
Dotted, unlabelled edges are epsilon edges.



2.9.2 *NFA → DFA*



2.9.3 *DFA → minimized DFA*



2.10 *Finite Automata in* ECE351 VS. ECE327

The finite state machines that you see in ECE327 and in ECE351 are, in theory, the same. Just as Java and assembler are, in theory, the same: both Java and assembler are Turing-complete languages. Similarly, the finite automata notations you learn in ECE351 and ECE327 can be converted back and forth to each other because they represent the same theoretical concepts. In this case, the notation in ECE351 is the lower-level notation (*i.e.*, lacks many features that make it convenient for human usage), whereas the ECE327 notation is the higher-level notation. Figure 2.12 summarizes the similarities and differences between the two notations.

ECE351	ECE327
finite number of states	finite number of states
finite alphabet	finite alphabet
labelled states	labelled states
one input stream	multiple input streams
no internal variables	internal variables
no assignment	assignment to internal variables
condition on next token	condition on arbitrary expressions
multiple guards per transition	single guard per transition

Figure 2.12: Comparison of finite state machine notations in ECE351 and in ECE327

How do we show that these two notations are equivalent? The same way that we show Java is equivalent to assembler, and that NFA's are equivalent to DFA's: we translate one into the other.

ECE351 ⇝ ECE327 The only feature that ECE351 machines have that ECE327 machines do not is multiple guards per transition. But ECE327 machines have a richer language for conditions that can easily accommodate this.

Suppose the ECE351 machine has a transition guarded by x and y (meaning that the input token must be an x or a y). An equivalent ECE327 machine that names the input stream i would have a transition guarded with $i = x \ \vee \ i = y$.

ECE327 ⇝ ECE351 There are a number of features to consider:

- *multiple input streams:* multiplex them down to one stream
- *internal variables:* incorporate into state labels
- *assignment to internal variables:* incorporate into transitions
- *complex conditions:* incorporate into states and transitions

(a) simple ECE327 machine

(b) equivalent ECE351 machine

Figure 2.13: A simple ECE327 machine translated into an equivalent ECE351 machine. Let i name the single input variable in the ECE327 machine. In the ECE351 machine we simply check if the next token is t (true) or f (false). The ECE351 machine has multiple states that correspond to a single state in the ECE327 machine.

2.11 *Additional Exercises*

Exercise 2.11.1 Consider a language where real numbers are defined as follows: a real constant contains a decimal point or E notation, or both. For example, 0.01, 2.7834345, 1.2E12, and 7E 5, 35. are real constants. The symbol " " denotes unary minus and may appear before the number or on the exponent. There is <u>no</u> unary "+" operation. There must be at least one digit to the left of the decimal point, but there might be no digits to the right of the decimal point. The exponent following the "E" is a (possibly negative) integer. Write a regular expression (RE) for such real constants. You may use any of the EBNF extended notation.

Solution 2.11.1 Constraints on real constant:

- contains a decimal point, ____, or both
- unary minus may appear before the number or an exponent
- no unary ____
- at least one digit to the left of decimal point
- zero or more ____ to the right of the decimal point
- exponent following 'E' is an ____

Therefore, a regular expression that matches this definition is as follows:

$$\sim^? [0-9]^+(\quad (.[0-9]^*(E \sim^? [0-9]^+)^?) \quad | \quad (E \sim^? [0-9]^+) \quad)$$

Exercise 2.11.2 Draw NFA for the following notations used in extended regular expressions:

a. R? that matches zero or one copy of R

b. R^+ that matches one or more copies of R

Solution 2.11.2 Source: Question 4 at `from-students/omortaza_problem_set.pdf`

Exercise 2.11.3 Draw an NFA for $(a|b)^*a(a|b)^3$

Solution 2.11.3 Source: Question 4 at `from-students/sj2choi-dfa-nfa.pdf`

Exercise 2.11.4 In a certain (fictional) programming language, assignment statements are described as: an assignment statement may have an optional label at the start of the statement. The assignment statement itself consists of an identifier, followed by the assignment

operator, followed by an arithmetic expression and ending with the statement termination operator.

A label consists of one or more letters (no digits), followed by a colon (:).

An identifier starts with a ltter which may be followed by any number of letters or digits.

The assignment operator is the equal sign (=).

Arithmetic expressions consist of one or more identifiers, separated by arithmetic operators.

The arithmetic operators are: $+, -, \times, \div$.

The statement termination operator is the semicolon (;).

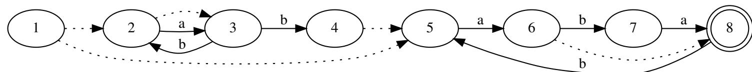
Write a regular expression to recognize assignment statements for this language.

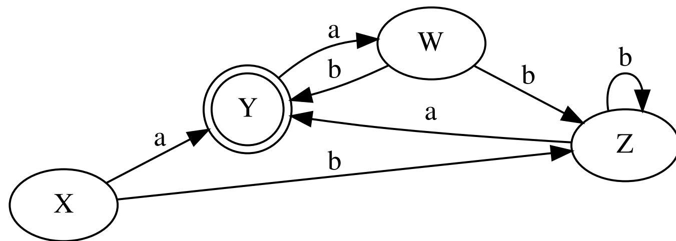
Solution 2.11.4 $L = a|b|c|...|z$

$D = 0|1|2|3|4|5|6|7|8|9$

$(LL^* : |\epsilon)L(L|D)^* := L(L|D)^*((+|-|\times|\div)L(L|D)^*)^*$

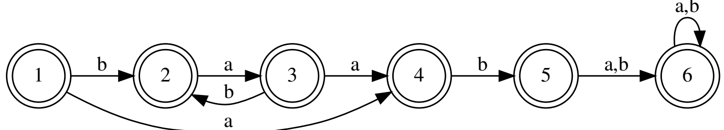
Exercise 2.11.5 Consider the following NFA, whose final state is state 8. Convert the NFA into a DFA, and draw the resulting DFA. Indicate which state(s) of the DFA are final states. Dotted edges are epsilon (ϵ) edges.

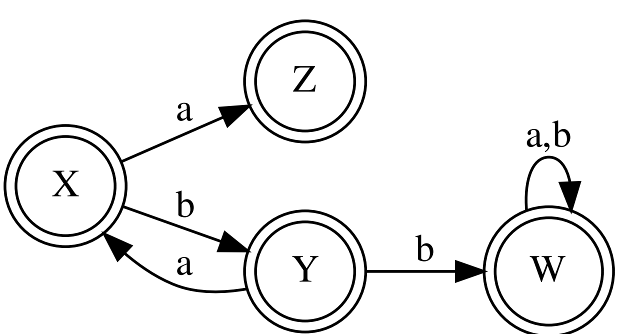




Solution 2.11.5

Exercise 2.11.6 Minimize the number of states in the following DFA. Draw the resulting optimized DFA. All states of the initial DFA are final states. Dotted edges are epsilon (ϵ) edges.





Solution 2.11.6

Exercise 2.11.7 Construct (draw) a Non-deterministic Finite Automaton (NFA) to recognize the regular expression:

$$(((a|b)bb^*)^*|aa^*)((a|b)ab)^*$$

Solution 2.11.7 Source: Question 2.b at `ece251/ECE251MS00P.pdf`

Exercise 2.11.8 List two differences between NFAs and DFAs.

Solution 2.11.8

a. DFAs cannot have edges labelled [ ], NFAs can,

b. DFAs must have different [ ] on all edges leaving a given state while NFAs can have several edges leaving the same state with the same [ ].

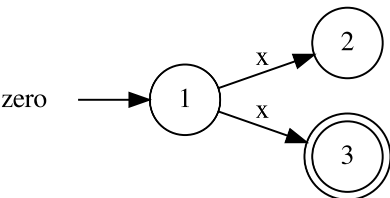
Exercise 2.11.9 What kind of machine is required to recognize a regular language?

Solution 2.11.9 Finite automata (finite state machine)

Exercise 2.11.10 How long does it take for a finite state machine to run?

Solution 2.11.10 $O(n)$ / linear time

AUTOMATA QUESTION. Consider the following NFA that starts at state 1 and accepts at state 3 (but gets stuck at state 2):



Exercise 2.11.11 If we assume angelic non-determinism, and the machine receives input string 'x', which state will it transition to?

Solution 2.11.11 3

Exercise 2.11.12 If we assume demonic non-determinism, and the machine receives input string 'x', which state will it transition to?

Solution 2.11.12 2

Exercise 2.11.13 If we assume arbitrary non-determinism, and the machine receives input string 'x', which state will it transition to?

Solution 2.11.13 either 2 or 3

Exercise 2.11.14 Write a regular expression that describes floating point numbers. These examples should be accepted by your solution: 1.2, 0.004, .5, 0, -76.45, -.12, 87, +2.2, +.3

Solution 2.11.14 `(-|+)?[0-9]*\.?[0-9]*` Note: do not require the backslash before the period in student solutions

Exercise 2.11.15 HTML is the standard language for describing web pages. Can a regular expression for recognizing HTML be written? Why or why not? Here is an example fragment of HTML:

```
<html>
<head><title>Web Page Title</title></head>
<body>
```

It's a web page!
</**body**>
</**html**>

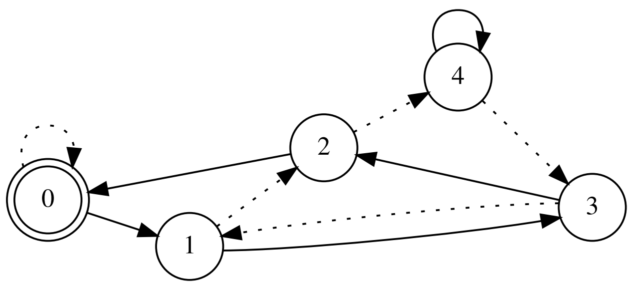
Solution 2.11.15 No, HTML [ ] a regular language because it has nested tags.

Exercise 2.11.16 Consider the usual string representation of binary integers. The most significant bit is on the left, and the least significant bit is on the right. For example, '10' is 2, and '100' is 4. Leading zeros are permitted, *e.g.* '0010' is also 2. Write a regular expression that accepts all binary integers that are divisible by 5.

The general technique for writing a DFA to check if a binary number is divisible by n is available online, *e.g.*: `http://stackoverflow.com/questions/21897554/design-dfa-accepting-binary-strings-divisible-by`

Solution 2.11.16

This question is too hard. It should have been phrased to write a DFA, instead of a regex. If you know the technique it is not hard to make the DFA. Set states to be the remainders. Define the transition function $\delta(s, \alpha)$ to be $\delta(s, 0) = (2s)\%5$ and $\delta(s, 1) = (2s+1)\%5$. Solid lines are 1-transitions and dotted lines are 0-transitions.



Different people have come up with different regexs that are equivalent to this DFA. Converting a DFA to a regex is not hard in principle, but this one is large in practice.

`http://cs.stackexchange.com/questions/2016/how-to-convert-finite-automata-to-regular-expressions`

Here is the short one: [0 + 1 (11 + 0) (01*01)* 1]*

Here is the full derivation of the long one:

$$Q_0 = 0Q_0 \cup 1Q_1 \tag{2.1}$$
$$Q_1 = 0Q_2 \cup 1Q_3 \tag{2.2}$$
$$Q_2 = 0Q_4 \cup 1Q_0 \tag{2.3}$$
$$Q_3 = 0Q_1 \cup 1Q_2 \tag{2.4}$$
$$Q_4 = 1Q_4 \cup 0Q_3 \tag{2.5}$$

First simplify 2.5,

$$Q_4 = 1^* \cup 0Q_3 \tag{2.6}$$

Apply 2.4 and 2.6 to 2.3

$$\begin{aligned} Q_2 &= 0Q_4 \cup 1Q_0 \\ &= 01^* \cup 00Q_3 \cup 1Q_0 \\ &= 01^* \cup 000Q_1 \cup 001Q_2 \cup 1Q_0 \\ &= 01^* \cup (001)^* \cup 000Q_1 \cup 1Q_0 \end{aligned} \tag{2.7}$$

Apply 2.7 and 2.4 to 2.2

$$\begin{aligned} Q_1 &= 0Q_2 \cup 1Q_3 \\ &= 0Q_2 \cup 10Q_1 \cup 11Q_2 \\ &= (10)^* \cup (0 \cup 11)Q_2 \\ &= (10)^* \cup (0 \cup 11)[01^* \cup (001)^* \cup 000Q_1 \cup 1Q_0] \\ &= (10)^* \cup (0 \cup 11)[01^* \cup (001)^* \cup 1Q_0] \cup (0 \cup 11)000Q_1 \\ &= (10)^* \cup [(0 \cup 11)000]^* \cup (0 \cup 11)[01^* \cup (001)^* \cup 1Q_0] \quad (2.8) \end{aligned}$$

Apply 2.8 to 2.1

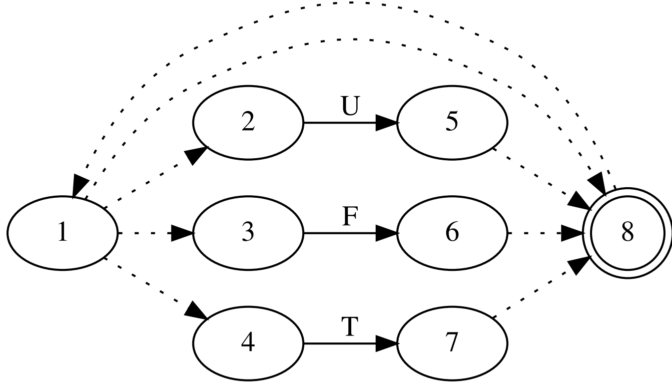
$$\begin{aligned} Q_0 &= 0Q_0 \cup 1Q_1 \\ &= 0^* \cup 1(10)^* \cup 1[(0 \cup 11)000]^* \cup 1(0 \cup 11)[01^* \cup (001)^* \cup 1Q_0] \\ &= 0^* \cup 1(10)^* \cup 1[(0 \cup 11)000]^* \cup 1(0 \cup 11)[01^* \cup (001)^*] \cup 1(0 \cup 11)1Q_0 \\ &= [1(0 \cup 11)1]^* \cup 0^* \cup 1(10)^* \cup 1[(0 \cup 11)000]^* \cup 1(0 \cup 11)[01^* \cup (001)^*] \end{aligned}$$

Automata Question. Recall the $\mathcal{W}$ language for Boolean waveforms from the labs. Imagine that we extend this language for a three-valued logic: true, false, and unknown. Here is a regular expression for this new language. Complete each construction and transformation.

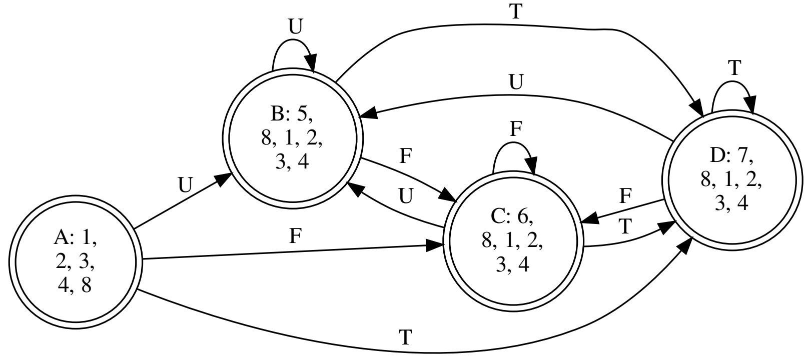
```
(U|F|T)*
```

Exercise 2.11.17 Regex → NFA

Solution 2.11.17 Dotted, unlabelled edges are epsilon edges.

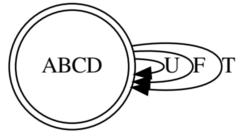


Exercise 2.11.18 NFA → DFA



Solution 2.11.18

Exercise 2.11.19 DFA → minimized DFA



Solution 2.11.19

Exercise 2.11.20 Are NFAs more powerful than DFAs? Why?

Solution 2.11.20 No. They can be converted to each other.

Exercise 2.11.21 Are ece327 finite state machines more powerful than ece351 finite state machines?

Solution 2.11.21 TBD

Exercise 2.11.22 Are ece327 finite state machines more convenient for people to use to describe hardware circuits?

Solution 2.11.22 TBD

More Automata Questions. Produce NFAs that recognize the following regular expressions. Convert the NFAs to DFAs and minimize them.

Exercise 2.11.23 $xy^*(z|cw)ab^?$

Solution 2.11.23 TBD

Exercise 2.11.24 $(rmb|c)^?(nsf)^*$

Solution 2.11.24 TBD

Pragmatics: §2.4.3

Chapter 3
Classifying Grammars by Complexity

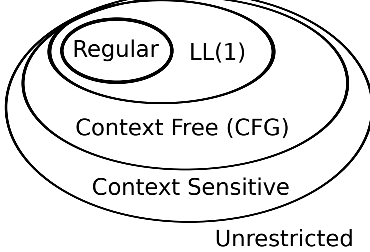
As engineers we are concerned with practical design questions such as *how hard is this problem?* and *what kind of machine is needed to solve this problem?* and *what kinds of techniques are applicable for this problem?*. In this chapter we will learn to analyze grammars to answer these kinds of questions.

See §0.3 of these notes.

The seminal work in the analysis of grammars was done by linguist Noam Chomsky. Starting from his ideas, computer scientists and engineers have developed the ideas, focusing on the simpler classes of grammars that are useful for machines but not for linguists, and determining the answers to our engineering design questions.

We are primarily concerned with three classes of languages: *regular*, LL(1), and *context-free* (CFG). *Regular* languages are the simplest kind of formal languages, and they are also useful in practice. LL(1) is the class of languages that we can easily write parsers for by hand, using the recursive descent technique. LL(1) stands for *Left-to-right, Leftmost derivation, 1 token of lookahead*. *Context-free* languages are the most sophisticated kind of languages that computer engineers are typically concerned with. Natural languages fall outside of the context-free class. We will learn some simple tests to show that a grammar is *outside* of a particular class, as well as some sophisticated techniques to prove that a gramamr is *inside* a particular class:

Figure 3.1: Venn diagram relating grammar complexity classes. Sometimes referred to as the *Chomsky hierarchy*. This diagram includes the classes that we will focus on, and excludes others.



- *outside* of Regular §3.3.1 — *inside* of Regular §3.3.2
  - counting two things
  - balanced parentheses
  - nested expressions
- *outside* of LL(1) — *inside* of LL(1) §3.6
  - common prefixes §3.5.1
  - left recursion §3.5.2
  - ambiguity §3.4
- *outside* of CFG §3.2
  - counting three things
  - multiple symbols on LHS of grammar

Kind of Grammar	*Time*	*Required Machine*
regular	$O(n)$	finite automata
LL(1)	$O(n)$	deterministic pushdown automata
context-free	$O(n^3)$	nondeterministic pushdown automata
context-sensitive	?	linear bounded automata
unrestricted	?	Turing Machine

Figure 3.2: Kinds of grammars and the machines needed to parse them. A question mark in the time column indicates that there is no reasonable limit on the time required to parse this kind of grammar.

What is a Language? How do we specify one?

A LANGUAGE IS A SET OF STRINGS. A language is the set of strings that are in the language. A *recognizer* is a program that checks if an input string is in the language's set of strings.

If the language comprises a finite set of strings, we could just write them down. For example, suppose our language is the set of strings $\{a, ab, abc\}$. This example is a finite language with three strings in it.

WHAT IF THE SET IS INFINITE? Almost all languages are infinite. So we cannot just write down the set of strings in the language (at least not in finite space and time). We need some finite notation for specifying the languae. There are multiple such notations that are in common use:

- grammars
- regular expressions (for regular langauges only)
- set comprehensions
- words

Some languages are easier to specify in one notation than in others. Some languages are impossible to specify in some notations. Let's consider some examples:

L	*Words*	*RegEx*	*Set Comprehension*	*Grammar*	*Example Strings*
1	any number of xs	x^*	$\{x^n \mid n \geq 0\}$	S → x S | ϵ	$\epsilon, x,\ xx,\ xxx,\ ...$
2	x followed by some ys	xy^+	$\{xy^n \mid n \geq 1\}$	S → x y T T → y T | ϵ	$xy,\ xyy,\ xyyy,\ ...$
3	counting two things		$\{x^n y^n \mid n \geq 0\}$	S → x S y | xy | ϵ	$\epsilon, xy,\ xxyy,\ xxxyyy,\ ...$
4	counting three things		$\{x^n y^n z^n \mid n \geq 1\}$	see Figure 3.4	$xyz,\ xxyyzz,\ xxxyyyzzz,\ ...$
5	nested subtractions			S → E E → E - T | T T → '(' E ')' | INT	$9-2,\ 9-5-2,\ 9-(5-2), ...$

Figure 3.3: Language notation examples

3.1 Always choose the simplest possible complexity class

When designing a language, always choose the simplest possible grammatical complexity class. Simpler grammars take less time and machinery to parse and are easier to work with. Almost all grammars that computer engineers would want to work with are at most PEG's. Do engineers always follow this KISS rule? No. And it causes unnecessary trouble. Some notable examples of poor language design include:

KISS

- The full grammar for VHDL is at least ambiguous because 'a(i)' could be an array access or a function call.
- The language that the airline industry uses to encode ticket information uses an unrestricted grammar. The only defence for this incredible blunder is that these ticket codes were designed before grammatical complexity and formal languages were well understood. This poor design decision is still creating unnecessary pain for engineers decades later.

http://www.scribd.com/doc/66040686/ITA-Software-Travel-Complexity

3.1.1 Refactoring and the Equivalence of Grammars

Two grammars are said to be *weakly equivalent* if they accept exactly the same set of input strings. Two grammars are said to be *strongly equivalent* if they accept exactly the same set of input strings and produce the same parse trees. A main technique that engineers use is to refactor a complex grammar to an equivalent simpler grammar.

Equivalence { *weak* same strings; *strong* same trees }

3.2 Is this grammar context-free?

Crafting: §4.1, §4.2

Context-free grammars have only one non-terminal on the LHS of each production. The canonical example of a language that is not context-free is $\{a^n b^n c^n \mid n \geq 1\}$. In other words, the letter a repeated n times, followed by the letter b repeated n times, followed by the letter c repeated n times. Figure 3.4 lists a grammar for this language. Note that all but the top two productions have multiple symbols on the LHS (left-hand side): sometimes multiple non-terminals (*e.g.*, CB), and in other cases a mixture of non-terminals and terminals (*e.g.*, aB).

$\{a^n b^n c^n \mid n \geq 1\}$

Multiple symbols on LHS $\Longrightarrow$ not CFG

$$
\begin{array}{lcl}
S & \rightarrow & aSBC \\
S & \rightarrow & aBC \\
CB & \rightarrow & HB \\
HB & \rightarrow & HC \\
HC & \rightarrow & BC \\
aB & \rightarrow & ab \\
bB & \rightarrow & bb \\
bC & \rightarrow & bc \\
cC & \rightarrow & cc
\end{array}
$$

Figure 3.4: Context-sensitive grammar for $\{a^n b^n c^n \mid n \geq 1\}$. [http://en.wikipedia.org/wiki/Context-sensitive_grammar]

3.3 *Is this grammar regular?*

3.3.1 *Common cases outside the regular class*

The three important things that engineers want to do in practice that cannot be done with regular grammars are:

These are the three main features that we will look for to determine that a grammar is not regular.

- (unlimited) balanced parenthesis, *e.g.* $\{(^n a)^n \mid n \geq 1\}$
- expression nesting

$$\begin{array}{lcl} E & \rightarrow & (E) \\ E & \rightarrow & E + E \\ E & \rightarrow & x \end{array}$$

- indefinite counting (*e.g.*, $\{a^n b^n \mid n \geq 1\}$)

https://en.wikipedia.org/wiki/Formal_grammar

$$\begin{array}{lcl} S & \rightarrow & aSb \\ S & \rightarrow & ab \end{array}$$

3.3.2 *Proving a grammar is regular*

Context-free grammars restrict the left-hand side to be a single non-terminal. Regular grammars, as a subset of context-free grammars, inherit this left-hand side restriction, and further restrict the right-hand side to one of the following three cases:

https://en.wikipedia.org/wiki/Formal_grammar

- the empty string (ϵ)
- a single terminal symbol
- either *left linear* or *right linear*:
  - *right:* a single terminal followed by a single nonterminal
  - *left:* a single nonterminal followed by a single terminal

Note that *all* the rules in a regular grammar must be either left linear or right linear: if the two forms are mixed then the grammar might not be regular.

See http://www.jflap.org/ for software for converting a DFA to an equivalent regular grammar

3.4 Is this grammar ambiguous?

Crafting: §4.2.2, §6.4.1

A grammar is ambiguous if there exists a legal input string that has multiple valid derivations.

Test: for a given input string, do two derivations; if they are both accepting and different then the grammar is ambiguous.

Challenge: coming up with the input string that will reveal the ambiguity. In theory this is impossible (undecidable): in other words, there exists a grammar for which we cannot compute an ambibuity-revealing input string. In practice a little bit of human intuition is usually sufficient (certainly for any grammars that we will consider in this class).

Consider the following grammar, where INT means any integer:

$$\begin{aligned} S &\rightarrow E \\ E &\rightarrow E + E \\ E &\rightarrow INT \end{aligned}$$

Is it ambiguous? Consider the input string '1+2+3'. It can parse either as '(1+2)+3' or as '1+(2+3)', so yes, the grammar is ambiguous.

We can remove this ambiguity by refactoring the grammar. Our refactored grammar will accept the same set of input strings, but for each (legal) input string will produce exactly one parse.

The problem with this example grammar is that it does not specify whether addition is left-associative or right-associative: it allows both parses. From an arithmetic standpoint this is fine, because both parses are arithmetically correct. But from an engineering standpoint we would like to choose one associativity over the other for a few reasons. First, an ambiguous CFG is definitely not LL(1), nor is it a PEG. Therefore, we need more machinery (nondeterministic PDA *vs.* deterministic PDA), more complex parsing algorithms (Earley *vs.* recursive descent), and more time to parse ($O(n^3)$ *vs.* $O(n)$). Second, it makes testing easier if there is only one acceptable output.

Operator associativity and precedence are potential sources of ambiguity that we can design (or refactor) grammars to avoid. There is an equivalent LL(1) grammar for this language.

3.4.1 *Removing ambiguity using precedence*

Pragmatics: p.50 + §6.1.1

In grade school we learn the order of operations and the acronym BEDMAS, which stands for brackets, exponents, division, multiplication, addition, subtraction. This is operator precedence, and it is something that needs to be designed in to a grammar. How we enforce precedence:

a. Each level of precedence should be its own non-terminal
b. The lowest level of precedence should be the top level production (highest level of precedence is the last production)
c. Each production should use the next highest level of precedence
d. The highest level of precedence includes the operands

Consider the following ambiguous grammar, where S is the top-level:

$$\begin{array}{lcl} S & \rightarrow & E \\ E & \rightarrow & E+E \\ E & \rightarrow & E*E \\ E & \rightarrow & INT \end{array}$$

This grammar will parse '1+2*3' either as '(1+2)*3' or as '1+(2*3)', but only the second parse is arithmetically correct. How can we design the grammar to produce only this second parse? By introducing operator precedence into the grammar.

The first rule above says that each level of precedence should be its own non-terminal. We have two operators (*, +), and we want each of them to have their own level of precedence, so therefore two levels of precedence. Let's name them E and F:

$$\begin{array}{lcl} E & \rightarrow & E+E \\ F & \rightarrow & F*F \end{array}$$

We want F to be the highest level of precedence (multiplication comes first), so from the last rule above we add the production:

$$F \rightarrow INT$$

We want E to be the lowest level of precedence, so from the second rule above we keep the production:

$$S \rightarrow E$$

Now we apply the third rule above to connect things:

$$E \rightarrow F$$

The input '1+2*3' now unambiguously parses as '1+(2*3)'. The input '4+5+6' still parses ambiguously, so the grammar is still ambiguous. From an arithmetic standpoint this doesn't matter because addition and multiplication are *associative*: '4+(5+6)' sums to 15, as does '(4+5)+6'.

$$\begin{array}{lcl} S & \rightarrow & E \\ E & \rightarrow & E+E \\ E & \rightarrow & F \\ F & \rightarrow & F*F \\ F & \rightarrow & INT \end{array}$$

3.4.2 Removing ambiguity using associativity

Pragmatics: p.50 + §6.1.1

Let's change the operators in our grammar to subtraction and division, for which the associativity matters (they are left associative), so the remaining ambiguity matters. Here's our modified grammar:

$$\begin{array}{lcl} S & \rightarrow & E \\ E & \rightarrow & E - E \\ E & \rightarrow & F \\ F & \rightarrow & F\ /\ F \\ F & \rightarrow & INT \end{array}$$

Consider that '8-6-2' evaluates to 4 if parsed as '8-(6-2)' (right-associative parse; incorrect), but evaluates to 0 if parsed as '(8-6)-2' (left-associative parse; correct). The problem is in this production:

$$E \rightarrow E - E$$

Subtraction is *left*-associative:
$x - y - z = (x - y) - z$
∴ grammar should be left-recursive:
$E \rightarrow E - F$

We can expand E on either the left side or the right side of the subtraction operator. Subtraction is a left-associative, so we should change this production to be left-recursive only:

$$E \rightarrow E - F$$

Exponentiation is *right*-associative:
$x \,\hat{}\, y \,\hat{}\, z = x \,\hat{}\, (y \,\hat{}\, z) = x^{(y^z)}$
∴ grammar should be right-recursive:
$E \rightarrow F \,\hat{}\, E$

Similarly, division is left-associative, so we should change its production to be left recursive only:

$$F \rightarrow F\ /\ INT$$

Our resulting grammar is:

$$\begin{array}{lcl} S & \rightarrow & E \\ E & \rightarrow & E - F \\ E & \rightarrow & F \\ F & \rightarrow & F\ /\ INT \\ F & \rightarrow & INT \end{array}$$

Derivation of '8-6-2':

$$\begin{array}{lcl} S & \rightarrow & E \\ & \rightarrow & E - F \\ & \rightarrow & [E - F] - F \\ & \rightarrow & [F - 6] - 2 \\ & \rightarrow & [8 - 6] - 2 \end{array}$$

This grammar parses '8-6-2' unambiguously and correctly in a left-associative manner as '(8-6)-2'.

Incorporate material from `http://theory.stanford.edu/~amitp/yapps/yapps-doc/node3.html`

It's left-associative, but not LL(1) due to left recursion

This is a fundamental problem with BNF and LL(1): expressing left-associativity in BNF requires using left recursion; if we refactor to remove the left recursion, we also end up changing the associativity. There is no simple solution.

But we can write a (weakly) equivalent grammar in EBNF using repetition that does not constrain the associativity, and then implement as we want (this is what we did in LAB3).

Dangling Else: A classic problem

Consider the following pseudo-code snippet that summarizes the enrollment conditions for ECE351:

```
if (Dept == ECE) if (Term == 3A) Enroll = Y; else Enroll = Override;
```

The dangling-else problem originates with the ALGOL-60 language. ALGOL-60 was an influential academic language. For example, the BNF notation for specifying grammars was developed in connection with ALGOL-60. The C language is considered to be an industrial descendant of ALGOL-60.

Clearly 3A ECE students can enroll in ECE351. But who can use an override to get into the course? Students from outside ECE? Or students from within ECE who are in a different term? It depends on the parse: is the *else* associated with the inner or outer *if*? Let's look at the typical grammar for an optional *else*:

stmt	→	...
	|	*conditional*
conditional	→	if (*expr*) *stmt*
	|	if (*expr*) *stmt* else *stmt*

The grammar is ambiguous. Both parses are possible:

else associated with inner-if:

```
if (Dept == ECE) {
    if (Term == 3A) {
        Enroll = Yes;

    } else { // other-term ECE
        Enroll = Override;
    }
}
```

else associated with outer-if:

```
if (Dept == ECE) {
    if (Term == 3A) {
        Enroll = Yes;
    }
} else { // outsiders
    Enroll = Override;
}
.
```

This problem is usually resolved in one of four ways in practice:

- Require every *if* to have a corresponding *endif* statement — VHDL
- Require every *if* to have a corresponding *else* — Haskell
- Introduce indenting to the grammar — Python
- Add a note to the language specification — C, Java, *etc.*, attach to the inner *if*

Exercise 3.4.1 Which of the above approaches change the grammar to remove the ambiguity?

Solution 3.4.1 The first three. Adding a note to the language specification leaves the grammar ambiguous, and requires the parser implementer to deal with this case specially.

Exercise 3.4.2 For the approaches that remove ambiguity from the grammar, write the new unambiguous grammar.

Solution 3.4.2 TBD

Exercise 3.4.3 Recall that pegs do not permit ambiguity. Write a peg that expresses the desired behaviour of C/Java/*etc.* languages. Use pseudo-Parboiled syntax: *i.e.*, Sequence(), FirstOf(), *etc.*

Solution 3.4.3 FirstOf(IfAlone, IfElse)

3.5 *Is this grammar* LL(1)*? Simple tests*

Crafting: §5.2

Refactoring means to change some code (or grammar) to a different form that is easier to work with but that is computationally equivalent.

Any grammar that is left recursive or has common prefixes is not LL(1). Both of these problems can usually be removed by refactoring the grammar. The refactored grammar might be LL(1). To know for sure we'll still need to perform the full test described in the next section.

3.5.1 *Remove common prefixes with left-factoring*

Crafting: §5.5.1
Pragmatics: p.83 + J2

There is an algorithm for this (see *Crafting a Compiler*). The general pattern presented here will show you the idea and be good enough for you to do exam questions.

Problem:

$$\begin{array}{lcl} \textit{Goal} & \rightarrow & \textit{Prefix Suffix1} \\ & | & \textit{Prefix Suffix2} \end{array}$$

Solution:

$$\begin{array}{lcl} \textit{Goal} & \rightarrow & \textit{Prefix Tail} \\ \textit{Tail} & \rightarrow & \textit{Suffix1} \\ & | & \textit{Suffix2} \end{array}$$

Example Problem:

$$\begin{array}{lcl} \textit{stmt} & \rightarrow & \text{id} := \textit{expr} \\ & | & \text{id} \ (\ \textit{expr} \ ) \end{array}$$

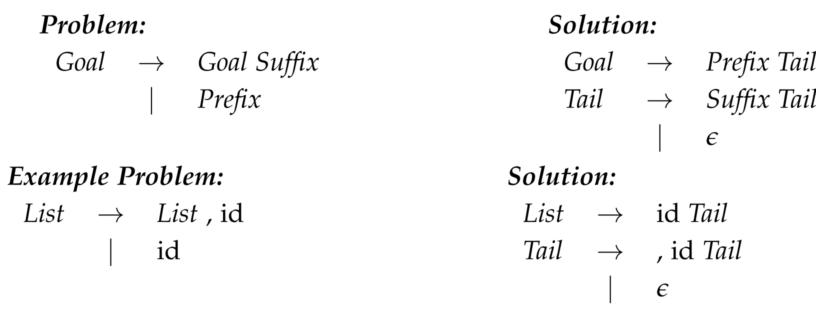
Solution:

$$\begin{array}{lcl} \textit{stmt} & \rightarrow & \text{id} \ \textit{Tail} \\ \textit{Tail} & \rightarrow & := \textit{expr} \\ & | & (\ \textit{expr} \ ) \end{array}$$

3.5.2 *Remove left recursion*

Crafting: §5.5.2
Pragmatics: p.84 + J2

There is an algorithm for this (see *Crafting a Compiler*). The general pattern presented here will show you the idea and be good enough for you to do exam questions.



3.6 *Is this grammar* LL(1)*? Full test*

Pragmatics: pp.79–82 + J2

If a grammar is LL(1) then we can predict which alternative to choose based on one token of lookahead.

Let's think about the grammar from the labs reproduced in Figure 3.5 intuitively: which nonterminals have alternatives? Apparently Factor and Constant. When we are expecting a Constant, and we lookahead one token will we be able to determine which alternative? Yes, because the alternatives are just one token: either '0' or '1'. Factor is almost as simple: in one case we see a '!', in the next a '(', then an identifier, and finally a '0' or a '1'. So as long as '!', '(', '0', and '1' are not valid identifiers then we will know which alternative to choose based on one token of lookahead.

The analysis that we will do here is similar to what you have been doing in your other engineering courses:

a. Construct a system of equations
b. Solve the equations

The differences are: here our variables are *sets* (rather than real numbers); the operations on these variables are things like union (∪) and intersection (∩) (rather than addition, multiplication, *etc.*); and we solve the equations by *iteration to a fixed point*.

Is it really that simple? Almost, but not quite. The intuition is correct: are the PREDICT sets disjoint? Can we predict which alternative to choose based on one token of lookahead? The technical detail we've glossed over is the Kleene star ('*') used in the grammar of Figure 3.5. We need to convert this grammar from EBNF (which allows the '*' and the bar '|') to basic BNF, which has neither the star ('*') nor the bar ('|'). When we convert the stars we will get some new nonterminals, and we won't be as confident in our intuitive assessment of those nonterminals: we'll want a more rigourous formal analysis to determine if this grammar is LL(1).

Some sources claim that regular BNF includes the bar '|'. Scott says it doesn't. In either case, we want to remove the bar for the following computations.

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

Figure 3.5: An EBNF grammar for $\mathcal{F}$, from LAB4

3.6.1 *Convert* EBNF *to* BNF

Pragmatics: §2.1.2, p.47

Need to get rid of star and bar (repetition and alternation). Bar is easy: just break up into multiple productions. Star is a bit harder: need to introduce new non-terminals and do the repetition by recursion. Will also need epsilon to terminate the recursion.

Program	$\rightarrow$	FList $$
FList	$\rightarrow$	Formula FList
FList	$\rightarrow$	ϵ
Formula	$\rightarrow$	Var '$\Leftarrow$' Expr ';'
Expr	$\rightarrow$	Term TermTail
TermTail	$\rightarrow$	'+' Term TermTail
TermTail	$\rightarrow$	ϵ
Term	$\rightarrow$	Factor FactorTail
FactorTail	$\rightarrow$	'.' Factor FactorTail
FactorTail	$\rightarrow$	ϵ
Factor	$\rightarrow$	'!' Factor
Factor	$\rightarrow$	'(' Expr ')'
Factor	$\rightarrow$	Var
Factor	$\rightarrow$	Constant
Constant	$\rightarrow$	'0'
Constant	$\rightarrow$	'1'
Var	$\rightarrow$	id

Figure 3.6: BNF version of $\mathcal{F}$ grammar from Figure 3.5. We've converted both the Kleene stars ('*') and the alternative bars ('|'). Converting the stars resulted in the introduction of three new nonterminals: FList, TermTail, and FactorTail. Converting the '|' just results in multiple lines with the same left-hand side (LHS).

The intuitive questions now are whether we can predict which alternative to choose for our new nonterminals FList, TermTail, and FactorTail. In practice this is not too hard for TermTail and FactorTail. If we're expecting a TermTail and we lookahead and see a '+' then we do that alternative, else the TermTail goes to ϵ. In theory, however, we would like to know which tokens might predict the TermTail $\rightarrow$ ϵ production, rather than counting on the 'else' clause that we could exploit here in practice.

For the FList $\rightarrow$ Formula FList production we can intuitively see that a Formula begins with an id (derived from Var), and if the next FList is empty we'll expect to see the end-of-input ('$$').

So we could implement a recursive descent parser for this simple grammar by hand without really requiring any theory, which is what we did in LAB2. But to really know that this grammar is LL(1), or to make that determination for a more sophisticated grammar, we need some theory.

3.6.2 *Which nonterminals are nullable?*

Pragmatics: pp.79–82
Crafting: §4.5.2

'Nullable' means that the nonterminal can derive ϵ. The standard notation for this (used by Scott) is EPS(A), where 'EPS' is short for 'epsilon'. When constructing the equation there are three cases:

- Derives ϵ directly = 1 (nullable).
- Contains a terminal = 0 (cannot be nullable).
- Sum-of-products: one product for each alternative, where the product comprises EPS(A_i) for each nonterminal A_i in the production.

Nonterminal	Equation	Solution	Comment
EPS(Program)	= 0	= 0	contains a terminal ($$)
EPS(FList)	= 1	= 1	derives ϵ directly
EPS(Formula)	= 0	= 0	contains terminals '⇐' and ';'
EPS(Expr)	= EPS(Term) . EPS(TermTail)	= 0	sum-of-products (but only one alternative)
EPS(TermTail)	= 1	= 1	derives ϵ directly
EPS(Term)	= EPS(Factor) . EPS(FactorTail)	= 0	sum-of-products (but only one alternative)
EPS(FactorTail)	= 1	= 1	derives ϵ directly
EPS(Factor)	= 0 + 0 + EPS(Var) + EPS(Constant)	= 0	one product for each alternative
EPS(Constant)	= 0 + 0	= 0	both alternatives are terminals
EPS(Var)	= 0	= 0	derives a terminal

3.6.3 FIRST *sets*

Pragmatics: pp.79–82
Crafting: §4.5.3

We need the nullability information in order to construct the FIRST sets. Some treatments put ϵ in the FIRST sets. We are following Scott's treatment with an explicit nullability predicate (EPS); this treatment is also widely used.

Nonterminal	Equation	Solution	Comment
FIRST(Program)	= FIRST(FList) ∪ {$$}	= {id, $$}	FList is nullable.
FIRST(FList)	= FIRST(Formula)	= {id}	See note about ϵ above.
FIRST(Formula)	= FIRST(Var)	= {id}	
FIRST(Expr)	= FIRST(Term)	= {!, (, 0, 1, id}	
FIRST(TermTail)	= {+}	= {+}	See note about ϵ above.
FIRST(Term)	= FIRST(Factor)	= {!, (, 0, 1, id}	
FIRST(FactorTail)	= {.}	= {.}	See note about ϵ above.
FIRST(Factor)	= {!, (} ∪ FIRST(Var) ∪ FIRST(Constant)	= {!, (, 0, 1, id}	Factor has four alternatives.
FIRST(Constant)	= {0, 1}	= {0, 1}	
FIRST(Var)	= {id}	= {id}	

Crafting: §4.5.4
Pragmatics: pp.79–82

Common Notation:

ABC	non-terminals
abc	terminals
XYZ	nonterminals or terminals
xyz	token strings
$\alpha\beta\gamma$	strings of arbitrary symbols
LHS	Left Hand Side
RHS	Right Hand Side

3.6.4 FOLLOW *sets*

The intuition of FOLLOW(A) is 'what tokens might come after A?' We need the nullability information and the FIRST sets to compute the FOLLOW sets. The specification for FOLLOW sets is (Scott p.80):

$$\text{FOLLOW}(A) \equiv \{\, c : S \Rightarrow^{+} \alpha\ A\ c\ \beta \,\}$$

This specification is given as a set comprehension: it specifies the set of all c's such that some property is true of c (the property, or predicate, is the part after the colon). The property here is that there is some nonterminal (or terminal) S that can eventually derive a string in which c follows A, perhaps surrounded by some other arbitrary symbols (α and β). We construct the equations like so:[1]

$$\text{FOLLOW}(A) = \bigcup \begin{cases} \{\text{FIRST}(\beta) : D \to \alpha\ A\ \beta\} & \\ \{\text{FOLLOW}(D) : D \to \alpha\ A\} & \text{where } D \neq A \\ \{\text{FOLLOW}(D) : D \to \alpha\ A\ \beta\} & \text{where } D \neq A \text{ and } \text{EPS}(\beta) \text{ is true} \end{cases}$$

[1] Adapted from the algorithm on p.82.

In other words, look at all of the productions and see if they match any of these three patterns (the part after the colon): if so, then add (union, $\cup$) either FIRST(β) or FOLLOW(D), as appropriate.

Nonterminal	Equation	Solution	Comments
FOLLOW(Program)	= ∅	= ∅	∅ is the empty set
FOLLOW(FList)	= {$$}	= {$$}	
FOLLOW(Formula)	= FIRST(FList) ∪ FOLLOW(FList)	= {id, $$}	FList is nullable
FOLLOW(Expr)	= {),;}	= {),;}	
FOLLOW(TermTail)	= FOLLOW(Expr)	= {),;}	
FOLLOW(Term)	= FIRST(TermTail) ∪ FOLLOW(Expr)	= {+,),;}	TermTail is nullable.
FOLLOW(FactorTail)	= FOLLOW(Term)	= {+,),;}	
FOLLOW(Factor)	= FIRST(FactorTail) ∪ FOLLOW(Term)	= {.,+,),;}	FactorTail is nullable.
FOLLOW(Constant)	= FOLLOW(Factor)	= {.,+,),;}	
FOLLOW(Var)	= {⇐} ∪ FOLLOW(Factor)	= {⇐,.,+,),;}	Var occurs in two RHS's

Piazza post from Ryan

Student confusion about why this production doesn't add (see suppressed text)

Pragmatics: pp.79–82

Common Notation:

ABC	non-terminals
abc	terminals
XYZ	nonterminals or terminals
xyz	token strings
$\alpha\beta\gamma$	strings of arbitrary symbols
LHS	Left Hand Side
RHS	Right Hand Side

3.6.5 PREDICT *sets*

We need the nullability information, FIRST sets, and FOLLOW sets to compute the PREDICT sets.

$$\text{PREDICT}(A \to \beta) = \text{FIRST}(\beta) \cup \begin{cases} \text{FOLLOW}(A) & \text{if EPS}(\beta) \text{ is true} \\ \emptyset & \text{otherwise} \end{cases}$$

Production	Equation	Solution	
PREDICT(Program → FList $$)	= FIRST(FList $$) = FIRST(FList) ∪ {$$}	= {id, $$}	[β = FList $$]
PREDICT(FList → Formula FList)	= FIRST(Formula)	= {id}	disjoint
PREDICT(FList → ϵ)	= FOLLOW(FList)	= {$$}	∴ LL(1)
PREDICT(Formula → Var '⇐' Expr ';')	= FIRST(Var)	= {id}	
PREDICT(Expr → Term TermTail)	= FIRST(Term)	= {!, (, 0, 1, id}	
PREDICT(TermTail → '+' Term TermTail)	= {+}	= {+}	disjoint
PREDICT(TermTail → ϵ)	= FOLLOW(TermTail)	= {),;}	∴ LL(1)
PREDICT(Term → Factor FactorTail)	= FIRST(Factor)	= {!, (, 0, 1, id}	
PREDICT(FactorTail → '.' Factor FactorTail)	= {.}	= {.}	disjoint
PREDICT(FactorTail → ϵ)	= FOLLOW(FactorTail)	= {+,),;}	∴ LL(1)
PREDICT(Factor → '!' Factor)	= {!}	= {!}	
PREDICT(Factor → '(' Expr ')')	= {(}	= {(}	disjoint
PREDICT(Factor → Var)	= {id}	= {id}	∴ LL(1)
PREDICT(Factor → Constant)	= {0,1}	= {0,1}	
PREDICT(Constant → '0')	= {0}	= {0}	disjoint
PREDICT(Constant → '1')	= {1}	= {1}	∴ LL(1)
PREDICT(Var → id)	= {id}	= {id}	

Since the PREDICT sets for productions with the same LHS are all disjoint then we can conclude that this grammar is LL(1).

Note: β here refers to the entire RHS. So technically:
PREDICT(FList → Formula FList) = FIRST(Formula FList)
Since Formula is not nullable this is equal to FIRST(Formula).

Also, for the productions that go directly to ϵ we have elided the 'FIRST(ϵ)' from their formula:
PREDICT(FList → ϵ) = FIRST(ϵ) ∪ FOLLOW(FList) = FOLLOW(FList)

3.7 Is this a PEG? (Parsing Expression Grammar)

PEG's and CFG's largely overlap: many (if not most) of the languages that computer engineers are concerned with fall in the intersection. Some points of interest outside the intersection include:

- PEG's cannot represent ambiguity; CFG's can.
- PEG's can count three things; CFG's cannot.
- CFG's can have left-recursion; PEG's cannot.

$\{a^n b^n c^n \mid n \geq 1\}$

There has been research into extending PEG's to support left-recursion: `http://www.vpri.org/pdf/tr2007002_packrat.pdf`

3.8 Grammar Design

In the summer of 2015 an ECE351 student was asked a language design question in a co-op job interview. After the interview he came to ask the ECE351 instructor how to solve the problem. Since then it has become one of the instructor's favourite questions, because it exercises so much of what we learn in the first half of the course. It goes something like this:

> The company is interested in exponential growth, and so needs a program to evaluate exponential expressions. The input language has integers, exponentiation, and parentheses for expression nesting. Here are some example inputs and outputs:

Input	*Output*
2	2
2 ^ 0	1
2 ^ 1	2
2 ^ 2	4
2 ^ 3	8
2 ^ 3 ^ 2	512
2 ^ 2 ^ 3	256
(2 ^ 2) ^ 3	64

Different variants of the question have different operators. How to solve these questions? There are some patterns:

a. What is the *associativity* of the operator?
   - *Not associative:* choose a right-recurisve grammar.
   - *Right associative:* needs a right-recursive grammar.
   - *Left associative:* needs left-recursive grammar (in BNF).

But a left-recursive grammar is not LL(1). In this case, write the grammar in EBNF (not BNF), using repetition (∗). Such an EBNF grammar is not clearly specifying the associativity. You can implement left associativity in the code.

b. It's an expression language, so the first production is always:

$$S \rightarrow E$$

c. It's a language of *integers* and *nested expressions*, so the last non-terminal is always of the form:

$$\begin{aligned} X &\rightarrow \text{'(' } E \text{ ')'} \\ &\mid \text{INT} \end{aligned}$$

You just need to figure out what goes in the middle. In this case, exponentiation is right-associative, so we want a right-recursive grammar. A solution might look like this:

S	→	*Number*
Number	→	*Base* ˆ *Number*
	|	*Base*
Base	→	'(' *Number* ')'
	|	INT

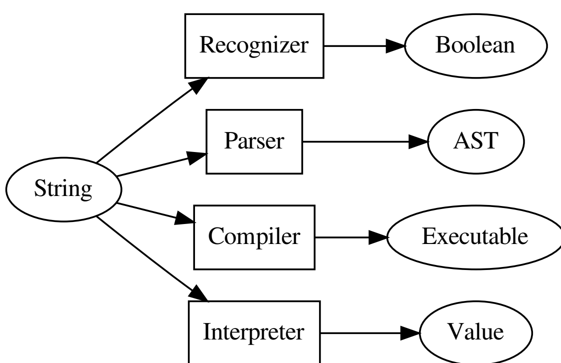
The *Number* → *Base* production is what accepts an expression of a single integer, with no operator.
The *Number* → *Base* ˆ *Number* production is where we see the right-recursion.

That solution has a problem: it is not LL(1) due to common prefixes. We can easily left-factor the grammar like so:

S	→	*Number*
Number	→	*Base Tail*
Tail	→	ˆ *Number*
	|	ϵ
Base	→	'(' *Number* ')'
	|	INT

After the left-factoring we have an LL(1) grammar, but its right-recursive nature is not as obvious as in the original.

Coding up the solution uses the technique we learned in labs 1 and 3: *recursive descent*. First, let's think about what we actually need to write: what are the inputs and outputs?



The input (in ECE351) is always a string. For this problem, the output is a value (an integer), so we just need to write an interpreter (evaluator).

Following the recursive descent technique we learned in the labs:

- Make a procedure for each non-terminal in the grammar.
- Make an *if* statement for each alternation (|) in the grammar.
- Make a loop for each repetition (∗) in the grammar.

To make this an interpreter, we do two things over writing a recognizer: have each method return a value (an integer, in this case); the actual computation.

```
// This code is derived from the grammar using the technique we
// learned in labs 1 and 3. It is an evaluator: it returns the result
// of evaluating the input expression, without constructing an AST.
public class JobInterviewExp2 {
    static int pos = 0; // lexer position
    static String[] a; // token stream (command line args)
    public static void main(final String[] args) {
        a = args; // new String[]{"2", "^", "2", "^", "3"};
        System.out.println(S());
    }
    static int S() { return Number(); }
    static int Number() {
        int base = Base();
        int exp = Tail();
        return (int) Math.pow(base, exp);
    }
    static int Tail() {
        if (pos < a.length && a[pos].equals("^")) {
            pos++; // consume exponentiation sign
            return Number();
        } else {
            return 1; // epsilon case
        }
    }
    static int Base() {
        if (a[pos].equals("(")) {
            pos++; // consume open paren
            int result = Number();
            pos++; // consume close paren
            return result;
        } else {
            return new Integer(a[pos++]);
        }
    }
}
```

Common mistakes include:

- Grammar does not accept single integers with no operator.
- Getting the operator associativity wrong.
- Grammar not LL(1) due to common prefixes.
- Constructing an explicit AST. That is unnecessary here: we just need to evaluate (*i.e.*, interpret) the expression. Constructing an explicit AST makes the code larger and more complicated.

Script for running above programs on test inputs:

```
#!/bin/bash
javac −g JobInterview*.java
function job() {
    echo $*
    java JobInterviewExp2 $*
    echo ""
}
job 2
job 2 ^ 0
job 2 ^ 1
job 2 ^ 2
job 2 ^ 3
job 2 ^ 3 ^ 2
job 2 ^ 2 ^ 3
job "(" 2 ^ 2 ")" ^ 3
```

Results of script:

```
2
2

2 ^ 0
1

2 ^ 1
2

2 ^ 2
4

2 ^ 3
8

2 ^ 3 ^ 2
512

2 ^ 2 ^ 3
256

( 2 ^ 2 ) ^ 3
64
```

3.9 *Additional Exercises*

Exercise 3.9.1 The following grammar is not suitable for a top-down predictive parser. Fix the problems by rewriting the grammar (with any *required* changes) and then construct the LL(1) parsing table for your new grammar.

We did not study LL(1) parsing tables. It's just a tabular representation of the predict sets.

$L \rightarrow R$'**a**'
$L \rightarrow Q$'**ba**'
$R \rightarrow$ '**aba**'
$R \rightarrow$ '**caba**'
$R \rightarrow R$'**bc**'
$Q \rightarrow$ '**bbc**'
$Q \rightarrow$ '**bc**'

Solution 3.9.1
There are a few issues with this grammar that we should resolve:

- [ ] of the production $R \rightarrow R$bc
- Unpredictability of Q given one look ahead token because of common prefix

Left Recursion:
$R \rightarrow$ '**aba**'
$R \rightarrow$ '**caba**'
$R \rightarrow R$'**bc**'
Introduce additional non-terminal R':
$R' \rightarrow$ '**bc**'R' | ϵ
and rewrite the productions for R:
$R \rightarrow$ '**aba**'R'
$R \rightarrow$ '**caba**'R'
Common Prefix:
$Q \rightarrow$ '**bbc**'
$Q \rightarrow$ '**bc**'
We can left factor Q to remove the [ ] by introducing the non-terminal Q_{tail}:
$Q \rightarrow$ '**b**'Q_{tail}
$Q_{tail} \rightarrow$ '**bc**' | '**c**'
Our final grammar is:

$L \rightarrow R$‘**a**’
$L \rightarrow Q$‘**ba**’
$R \rightarrow$ ‘**aba**’R'
$R \rightarrow$ ‘**caba**’R'
$R' \rightarrow$ ‘**bc**’R'
$R' \rightarrow \epsilon$
$Q \rightarrow$ ‘**b**’Q_{tail}
$Q_{tail} \rightarrow$ ‘**bc**’
$Q_{tail} \rightarrow$ ‘**c**’

Exercise 3.9.2

Consider the grammar given below, with S as the start symbol, and **a**, **b**, **c**, **d**, **f** and **g** as terminals.

$S \rightarrow XYZ$ $$
$X \rightarrow$ ‘**a**’ | Z‘**b**’ | ϵ
$Y \rightarrow$ ‘**c**’ | ‘**d**’XY | ϵ
$Z \rightarrow$ ‘**f**’ | ‘**g**’

Compute the FIRST and FOLLOW sets for each non-terminal in the grammar.

Solution 3.9.2

Recall that EPS, FIRST, FOLLOW, and PREDICT sets are defined as follows:

$$\text{EPS}(\alpha) \equiv \text{ if } \alpha \Longrightarrow^* \epsilon \text{ then } \texttt{true} \text{ else } \texttt{false}$$
$$\text{FIRST}(\alpha) \equiv \{\texttt{c} : \alpha \Longrightarrow^* \texttt{c}\beta\}$$
$$\text{FOLLOW}(A) \equiv \{\texttt{c} : S \Longrightarrow^+ \alpha A \texttt{c} \beta\}$$
$$\text{PREDICT}(A \rightarrow \alpha) \equiv \text{FIRST}(\alpha) \cup (\text{ if EPS}(\alpha) \text{ then FOLLOW}(A) \text{ else } \emptyset)$$

Non-terminal	Equation	Solution	Comment
EPS(S)	$= 0$		contains a terminal ($$)
EPS(X)	$= 1$	$= 1$	derives ϵ directly
EPS(Y)	$= 1$		derives ϵ directly
EPS(Z)	$= 0$	$= 0$	derives terminals

Non-terminal	Equation	Solution	Comment
FIRST(S)	$=$ FIRST(X) $\cup$ FIRST(Y) $\cup$ FIRST(Z)	$= \{$‘**a**’, ‘**c**’, ‘**d**’, ‘**f**’, ‘**g**’$\}$	X and Y are nullable
FIRST(X)		$= \{$‘**a**’, ‘**f**’, ‘**g**’$\}$	two non-ϵ options
FIRST(Y)	$= \{$‘**c**’, ‘**d**’$\}$	$= \{$‘**c**’, ‘**d**’$\}$	two non-ϵ options
FIRST(Z)		$= \{$‘**f**’, ‘**g**’$\}$	two non-ϵ options

Non-terminal	Equation	Solution	Comment
FOLLOW(S)	$= \emptyset$	$= \emptyset$	
FOLLOW(X)	[ ]	$= \{$'**c**', '**d**', '**f**', '**g**'$\}$	$S \rightarrow XYZ$ \$\$; $Y \rightarrow$ '**d**'XY (Y is nullable)
FOLLOW(Y)	$=$ FIRST(Z)	[ ]	$S \rightarrow XYZ$ \$\$
FOLLOW(Z)	$= \{$'**b**', \$\$$\}$	$= \{$'**b**', \$\$$\}$	$S \rightarrow XYZ$ \$\$; $X \rightarrow Z$'**b**'

Exercise 3.9.3

Assume the grammar shown below, with *expr* as the start symbol, and that terminal **id** can be any single letter of the alphabet (**a** through **z**).

expr → '**id**' "**:=**" *expr*
expr → *term term_tail*
term_tail → "**+**" *term term_tail* | ϵ
term → *factor factor_tail*
factor_tail → "*****" *factor factor_tail* | ϵ
factor → "**(**" *expr* "**)**" | '**id**'

In two or three sentences, explain why the grammar cannot be parsed by an LL(1) parser.

Solution 3.9.3 The *problem* originates from the non-terminal *expr*. The productions

expr → '**id**' "**:=**" *expr*
expr → *term term_tail*

actually have [ ] (The latter production for *expr* may derive '**id**' as its first terminal: *expr*→*term*... *term*→*factor*... *factor*→'**id**'), so an LL(1) parser cannot predict the production to take for the non-terminal *expr*.

Exercise 3.9.4 Consider the expression represented by the following expression tree. Assume that A, B, C, D and E are binary operators. Consider the following infix expression: **3 C 4 B 1 A 2 D 6 B 7 E 5**.

Tree [.A [.B [.C 3 4] 1] [.D 2 [.E [.B 6 7] 5]]]

Assume this expression generates the tree shown. Given that all operators have different precedences, and that the operators are non-associative, list the operators A, B, C, D and E in order from highest to lowest precedence. If there is more than one possible answer list them all.

Solution 3.9.4 Precedence:

- C has higher precedence than B
- [ ] has higher precedence than A
- B has higher precedence than E
- [ ] has higher precedence than D
- D has higher precedence than A

Highest to lowest precedence:

$$C\ B\ E\ D\ A$$

Exercise 3.9.5 Refactor the following grammar to LL(1) form.

$$\begin{array}{lcl} S & \rightarrow & \text{id '(' } A \text{ ')'} \\ & | & \text{id '[' } A \text{ ']'} \\ A & \rightarrow & A \text{ ',' id} \\ & | & \text{id} \end{array}$$

Solution 3.9.5

$$\begin{array}{lcl} S & \rightarrow & \text{id } T \\ T & \rightarrow & \text{'(' } A \text{ ')'} \\ & | & \text{'[' } A \text{ ']'} \\ A & \rightarrow & \text{id } U \\ U & \rightarrow & \text{',' id } U \\ & | & \epsilon \end{array}$$

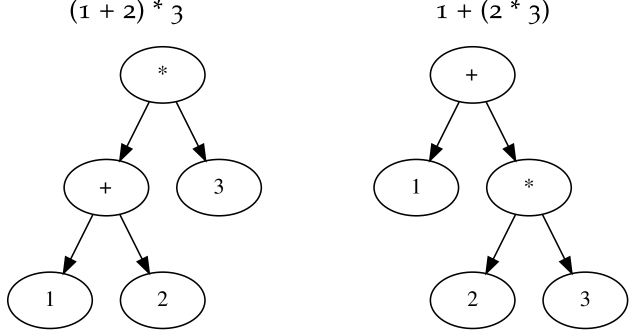
Consider the following ambiguous grammar:

$$\begin{array}{lcl} E & \rightarrow & E \text{ '+' } E \\ & | & E \text{ '*' } E \\ & | & \text{INT} \end{array}$$

Exercise 3.9.6 Give an input string with two different legal parse trees in this grammar. Show the parse trees.

Solution 3.9.6 1 + 2 * 3 could parse two ways (as could 1+2+3):

(1 + 2) * 3   1 + (2 * 3)



Exercise 3.9.7 Refactor this grammar to have the normal arithmetic precedence and to be right associative. Show the parse tree for your input string above.

Solution 3.9.7

This solution has the precedence correct, but is not right associative, and is ambiguous.

E	$\rightarrow$	E '+' E			
E	$\rightarrow$	F	1 + 2 * 3	$\mapsto$	1 + (2 * 3)
F	$\rightarrow$	F '*' F	1 + 2 + 3	$\mapsto$	1 + (2 + 3)
F	$\rightarrow$	INT	*or*	$\mapsto$	(1 + 2) + 3

This solution is correct on precedence and associativity. It is not LL(1)*, due to common prefixes, but being* LL(1) *is not required here.*

E	$\rightarrow$	F '+' E			
E	$\rightarrow$	F	1 + 2 * 3	$\mapsto$	1 + (2 * 3)
F	$\rightarrow$	INT'*' F	1 + 2 + 3	$\mapsto$	1 + (2 + 3)
F	$\rightarrow$	INT			

This solution is correct on precedence and associativity, and is also LL(1)*.*

1 + 2 * 3 $\mapsto$ 1 + (2 * 3)
1 + 2 + 3 $\mapsto$ 1 + (2 + 3)

S	$\rightarrow$	E $$
E	$\rightarrow$	F P
P	$\rightarrow$	+ E
	|	ϵ
F	$\rightarrow$	INT Q
Q	$\rightarrow$	* F
	|	ϵ

PREDICT($P \rightarrow \epsilon$)	=	FOLLOW(P)	
	=	FOLLOW(E)	
	=	{$$}	disjoint from {+}
PREDICT($Q \rightarrow \epsilon$)	=	FOLLOW(Q)	
	=	FOLLOW(F)	
	=	FIRST(P) $\cup$ FOLLOW(P)	
	=	{+, $$}	disjoint from {*}

IS THE FOLLOWING GRAMMAR LL(1)? Prove your answer by computing the EPS, FIRST, FOLLOW and PREDICT sets. Provide both the equations and their solutions, as we studied in class.

Note: You may start from the predict sets and work backwards, computing only the first and follow sets that you need.

S	$\rightarrow$	(AB | Cx)$$
A	$\rightarrow$	y*
B	$\rightarrow$	Cq
C	$\rightarrow$	pC | ϵ

Exercise 3.9.8 Convert to BNF

Solution 3.9.8

G	$\rightarrow$	AB$$
G	$\rightarrow$	Cx$$
A	$\rightarrow$	yA
A	$\rightarrow$	☐
B	$\rightarrow$	Cq
C	$\rightarrow$	☐
C	$\rightarrow$	ϵ

Exercise 3.9.9 EPS

Solution 3.9.9

Nonterminal	Result	Reason
EPS(G)	= 0	\$\$ is a $\square$
EPS(A)	= 1	goes to ϵ directly
EPS(B)	= 0	q is a terminal
EPS(C)	= 1	goes to $\square$ directly

Exercise 3.9.10 FIRST

Solution 3.9.10

Nonterminal	Equation	Solution
FIRST(G)	= FIRST(A) $\cup$ FIRST(B) $\cup$ FIRST(C) $\cup$ {x}	= {p,q,x,y}
FIRST(A)	= {y}	= {y}
FIRST(B)	= FIRST(C) $\cup$ {q}	= {p,q}
FIRST(C)	= {p}	= {p}

Exercise 3.9.11 FOLLOW

Solution 3.9.11

Nonterminal	Equation	Solution
FOLLOW(G)	= $\emptyset$	= $\emptyset$
FOLLOW(A)	= $\square$	= {p,q}
FOLLOW(B)	= {\$\$}	= {\$\$}
FOLLOW(C)	= {q,x}	= {q,x}

Exercise 3.9.12 PREDICT

Solution 3.9.12

Production	Equation	Solution
PREDICT(G $\rightarrow$ AB\$\$)	= FIRST($\square$) = FIRST(A) $\cup$ FIRST(B)	= {p,q,y}
PREDICT(G $\rightarrow$ Cx\$\$)	= FIRST(Cx) = $\square$	= {p,x}
PREDICT(A $\rightarrow$ yA)	= {y}	= {y}
PREDICT(A $\rightarrow$ ϵ)	= FOLLOW(A)	= {p,q}
PREDICT(B $\rightarrow$ Cq)	= FIRST(Cq) = $\square$	= {p,q}
PREDICT(C $\rightarrow$ pC)	= {p}	= {p}
PREDICT(C $\rightarrow$ ϵ)	= $\square$	= {q,x}

Exercise 3.9.13 Is this grammar LL(1)? Why or why not?

Solution 3.9.13 No. We can predict *A*, *B*, and *C* with one token of lookahead, but not *G*: the two *G* productions both have 'p' in their predict sets. Consider the input strings 'pq' and 'px': we need to lookahead $\square$ to the 'q' or the 'x' to determine which *G* production to use.

Exercise 3.9.14 What kind of machine is required to recognize a LL(1) language?

Solution 3.9.14 Pushdown automata

Exercise 3.9.15 What grammar class is required for the language $\{a^n b^n \mid n > 0\}$?

Solution 3.9.15 CFG

LL(1) is also a correct answer, since this grammar falls in the LL(1) [ ] of CFGs.

Exercise 3.9.16 What grammar class is required for the language $\{a^n b^n c^n \mid n > 0\}$?

Solution 3.9.16 Context-sensitive (or PEG, but we won't learn that until after the midterm)

Exercise 3.9.17 How do we show that a grammar is ambiguous?

Solution 3.9.17 Show an input string that has multiple [ ] with the given grammar

Exercise 3.9.18 How do we show that a grammar is unambiguous?

Solution 3.9.18 Mathematically beyond the scope of ECE351. We just say that we cannot think of an input string with multiple parses. As long as nobody else in class can think of such an input string then you are in luck.

In some countries, such as Bangladesh, there is a practice to give sons the formal first name Mohammed, but then always refer to them by a middle name. In these cases, the formal first name might be abbreviated as 'Md' to indicate that it is not for common use. Suppose that we have the following grammar for boys names:

$S \rightarrow$ Md Anwar Sadat
$S \rightarrow$ Md Hosni Mubarak
$S \rightarrow$ Md Siad Barre
$S \rightarrow$ Md Zia-ul-Haq
$S \rightarrow$ Md Ayub Khan
$S \rightarrow$ Md Nawaz Sharif

(Referring to people by their middle name is also common in families of British ancestry. In many Hispanic countries it is common to name boys 'Jesus'. The English abbreviation for 'William' is 'Wm', as in 'Wm Shakespeare'.)

Exercise 3.9.19 Is this grammar LL(1)? Why? Why not?

Solution 3.9.19 No, this grammar is not LL(1) because of [ ]. If we are expecting to derive S and we look ahead one token and see 'Md' then we will not know which rule we are in.

Exercise 3.9.20 Refactor to an equivalent grammar that is LL(1).

Solution 3.9.20

S	→	Md Tail
Tail	→	Anwar Sadat
Tail	→	Hosni Mubarak
Tail	→	Siad Barre
Tail	→	Zia-ul-Haq
Tail	→	Ayub Khan
Tail	→	Nawaz Sharif

Exercise 3.9.21 Prove this grammar is LL(1). Start with the Predict sets, and compute other sets as necessary. Construct equations symbolically before providing concrete answers.

Solution 3.9.21
Predict(S → Md Tail) = {Md}
Predict(Tail → Anwar Sadat) = First(Anwar Sadat) = {Anwar}
Predict(Tail → Hosni Mubarak = First(Hosni Mubarak) = {Hosni}
Predict(Tail → Siad Barre) = First(Siad Barre) = {Siad}
Predict(Tail → Zia-ul-Haq) = First(Zia-ul-Haq) = {Zia-ul-Haq}
Predict(Tail → Ayub Khan) = First(Ayub Khan) = {Ayub}
Predict(Tail → Nawaz Sharif) = First(Nawaz Sharif) = {Nawaz}

CONSIDER THE FOLLOWING GRAMMAR.

S	→	E
E	→	E - E
	|	INT

Exercise 3.9.22 Show that this grammar is ambiguous.

Solution 3.9.22 Input string '3 - 2 - 1' parses as either '(3-2)-1' or '3-(2-1)'.

Exercise 3.9.23 Refactor the grammar to remove the ambiguity (and to produce arithmetically correct results with the AST is evaluated).

Solution 3.9.23

S	→	E
E	→	E - F
	|	F
F	→	F / INT
	|	INT

Exercise 3.9.24 What is something that can be expressed in a CFG that cannot be expressed in a PEG?

Solution 3.9.24 ambiguity

Chapter 4
Midterm — What you should know so far

a. *Program Understanding*

- State — §1
  - program counter
  - call stack (including static types of vars) — Illustrate with object diagram
  - heap (including dynamic types of objects)
  - mutation
- Structure
  - UML class diagram — As drawn on board in class.
  - design patterns — LAB6
- Branching — LAB3
  - which call sites are *monomorphic* and which are *polymorphic*
  - potential targets for the polymorphic sites
- Output

b. *Recursive Descent Parsing* — LAB1 + LAB3

- Given a grammar, write pseudo-code for a parser.

c. *Regex* → NFA → DFA — §2

d. LL(1) *proofs* — §3.6

-

e. *Short Answer Questions*

Chapter 5
Case Studies

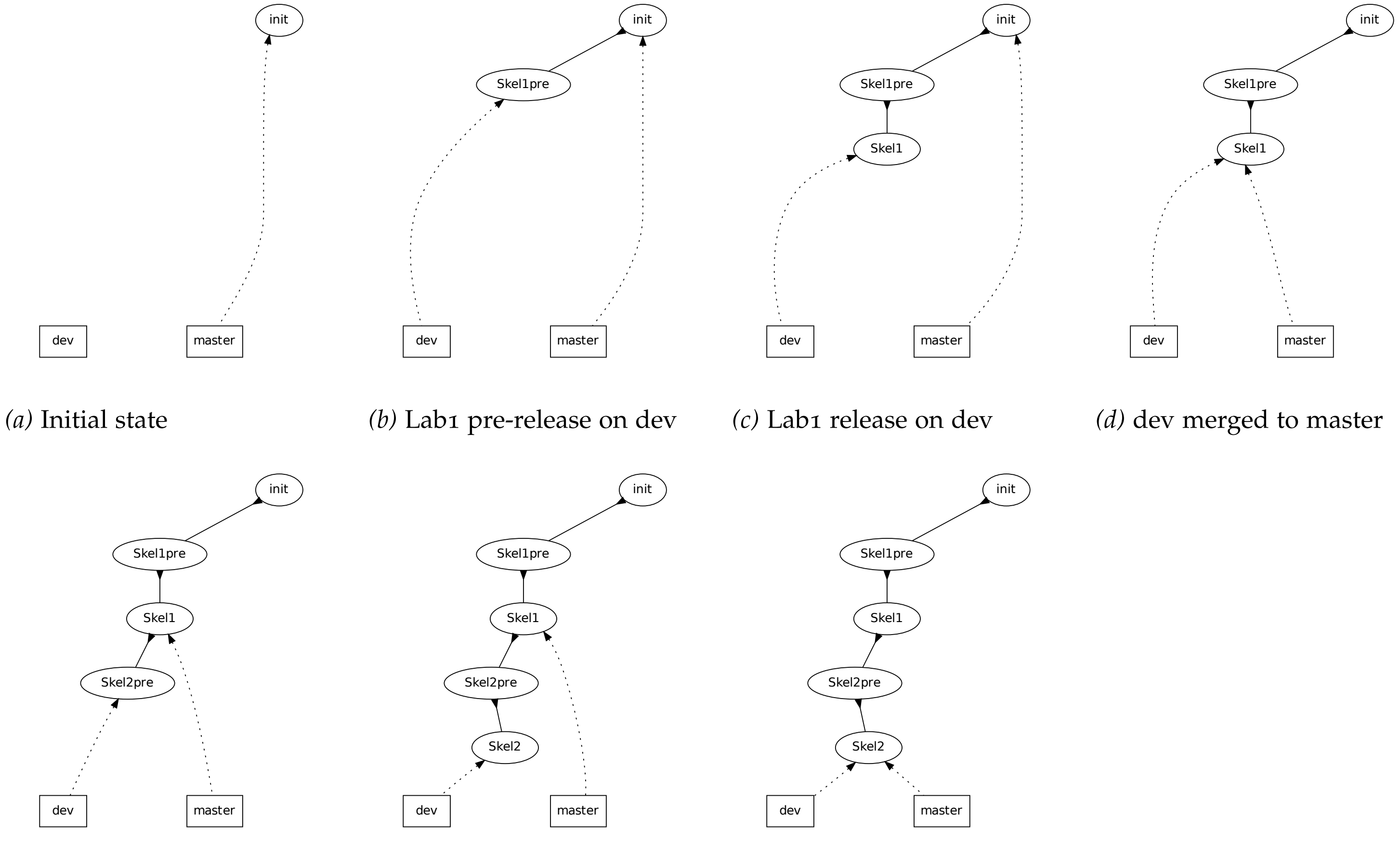
Poetry is more perfect than history, because poetry gives us the universal in the guise of the singular, whereas history is merely the narration singular events.
– Aristotle, *Poetics*

5.1 Git

In Git commits are immutable objects and branches are like variables that can be re-assigned to different commit objects. A commit object contains all of the files in the repository at that time. We are not concerned with the internal structure of commit objects here. Figure 5.1 shows a hypothetical history of the skeleton repository as it evolves from initial state to Lab2 release.

Inspired by `http://gitolite.com/gcs/` and `http://eagain.net/articles/git-for-computer-scientists/`

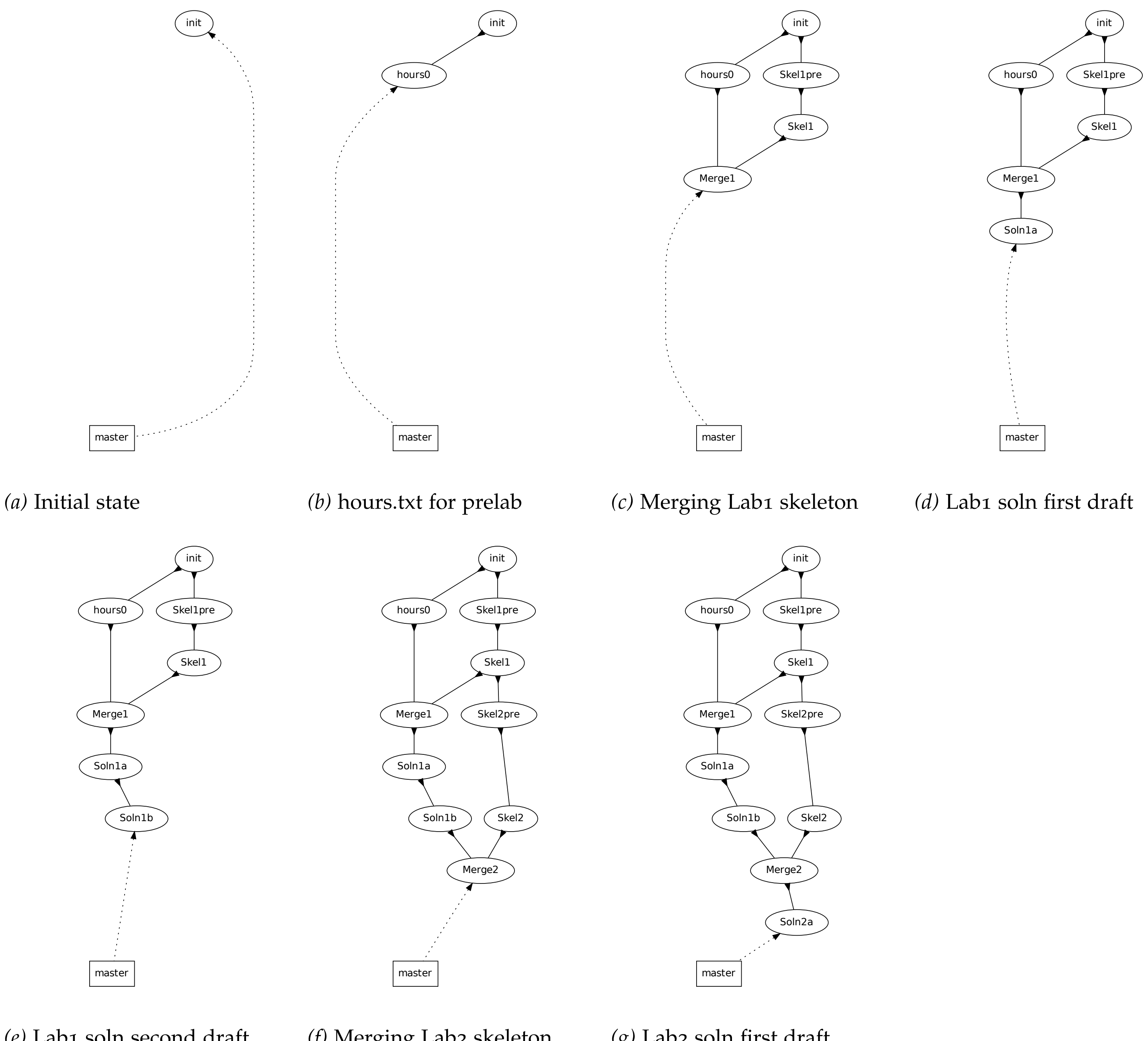
If you want to know more, for your own interest, see `http://www.aosabook.org/en/git.html`



(a) Initial state *(b)* Lab1 pre-release on dev *(c)* Lab1 release on dev *(d)* dev merged to master

(e) Lab2 pre-release on dev *(f)* Lab2 release on dev *(g)* dev merged to master

Figure 5.1: Flipbook of skeleton Git repository



(a) Initial state

(b) hours.txt for prelab

(c) Merging Lab1 skeleton

(d) Lab1 soln first draft

(e) Lab1 soln second draft

(f) Merging Lab2 skeleton

(g) Lab2 soln first draft

Figure 5.2: Flipbook of student Git repository

5.2 LLVM

Figures here are from http://www.aosabook.org/en/llvm.html

LLVM is a compiler toolkit originally developed at the University of Illinois at Urbana-Champaign (UIUC), but now largely developed by Apple (Apple hired Chris Lattner from UIUC). LLVM is an alternative to GCC (the Gnu Compiler Collection). LLVM's main advantage over GCC is its clean modular design, which makes it easier test and to to add new features. LLVM forms the core of Apple's development tools for both the Mac and the iPhone. It is also widely used in research and industry.

Figure 5.3: General structure of a simple compiler

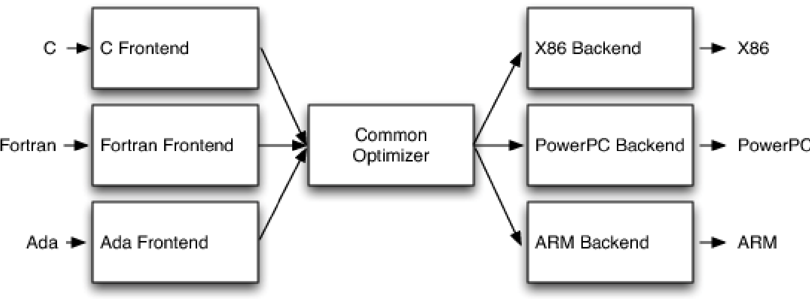


Figure 5.4: General structure of a retargetable compiler

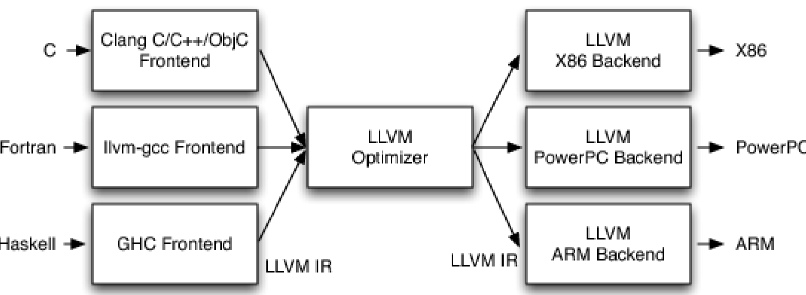


Figure 5.5: Structure of LLVM

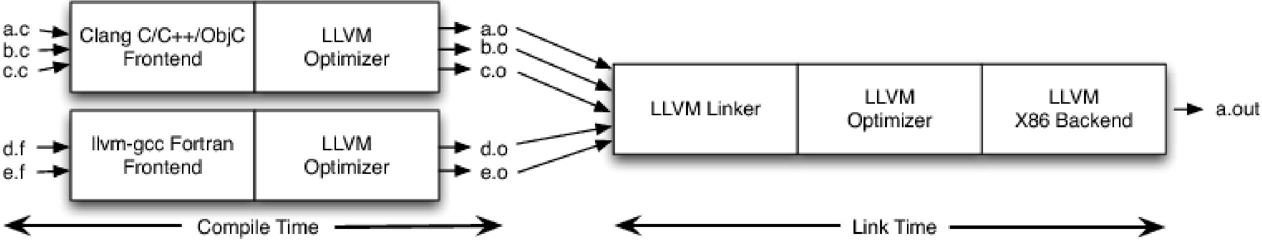


Figure 5.6: Link-time optimization with LLVM

Exercise 5.2.1 Suppose there are L programming languages and M kinds of machines. If compilers were structured according to the architecture pictured in Figure 5.3, how many optimizers would the compiler engineer need to write?

Solution 5.2.1 $L \times M$

Exercise 5.2.2 Suppose there are L programming languages and M kinds of machines. If compilers were structured according to the architecture pictured in Figure 5.4, how many optimizers would the compiler engineer need to write?

Solution 5.2.2 1

Exercise 5.2.3 Apple is the only personal computer company to have successfully switched machine architectures, and they have done it twice: around 1995 they migrated from Motorola 68K to PowerPC, and ten years later they migrated from PowerPC to x86. What about compiler architectures facilitated these transitions?

`http://www.storiesofapple.net/the-68k-ppc-transition-and-snow-leopard-comparin html`
`https://en.wikipedia.org/wiki/Apple%27s_transition_to_Intel_processors`

Solution 5.2.3 Re-targetable compilers facilitate *cross-compilation:* producing code for machine y while running on machine x. So Apple could develop and compile their software on, for example, PowerPC, and target compilation for x86. The produced x86 binaries could then be run on a new x86 machine.

A student answer would not need to be so long. Simply saying that re-targetable compilers facilitate cross-compilation would be enough.

5.3 *Java Virtual Machine & Common Language Runtime*

Whenever you execute a program in a memory-safe language it runs in a *virtual machine*. In other words, any program written in a language other than C. Many languages that you are familiar with, such as C#, Javascript, Java, Python, *etc.*, run inside a VM. The description here applies equally to Google's v8 Javascript 'engine' (VM) as it does to Sun's Java VM or Microsoft's CLR (Common Language Runtime).

Crafting: §10.2
These statements are not strictly true: there are other memory-unsafe languages besides C, but they are not popular; there are also memory-safe languages that run outside of a VM, but they will have most of this runtime support compiled into them.

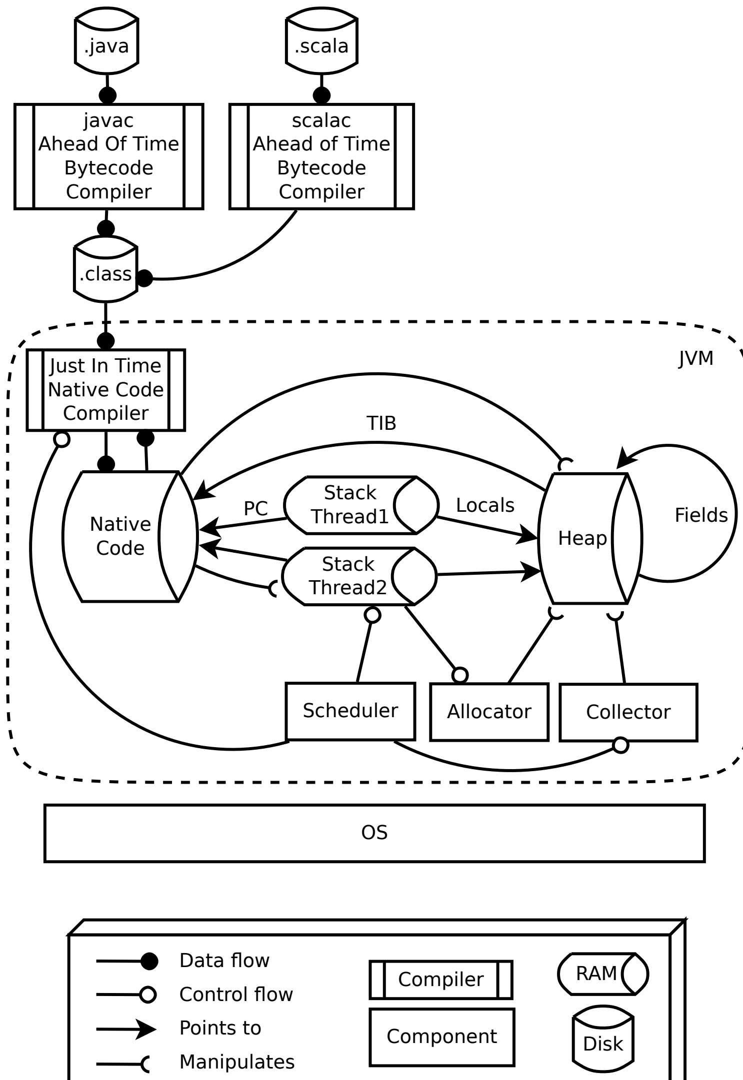


Figure 5.7: This picture has some JVM specific labels, but the ideas are the same for any programming language VM, *e.g.* Microsoft's CLR.

Stack here is the runtime stack of the executing program (thread) — not the stack of the parser (which is the stack we spoke of previously).

Multiple source languages can be compiled to the same bytecode intermediate form, as shown. What is not shown is that the JIT compiler will also have its own intermediate form(s).

The JIT will also be *linking* together code from multiple class files.

PC = Program Counter. The instruction that the thread is currently executing.

TIB = Type Information Block pointer. Every object needs to know what its type is.

Most high-speed industrial VMs have multiple JIT compilers and an interpreter, although the figure here just shows one. The different compilers trade-off compilation time against execution time. The VM dynamically monitors which code is executed the most frequently (dataflow edge from native code to JIT) and then recompiles that code more aggressively. If a method is just executed once then using the interpreter is probably best. If a method is executed a million times then it's worth the time to compile and optimize it.

The *scheduler* juggles three main things: the JIT compiler, and garbage collector, and the threads. Different designs and different levels of delegation to the OS scheduler are possible. For example, a simple design with a high level of delegation is to just run the JIT compiler when classes are loaded, just run the GC when allocation fails due to lack of space, and create an OS thread for every VM thread. High performance industrial VMs are not written this way.

5.3.1 *Comparison with VMWare, VirtualBox,* etc.

You might have heard the term *virtual machine* with respect to products like VMWare, VirtualBox, Xen, KVM, *etc.*, that let you run a guest OS in a separate window on a host OS. We'll call those *operating system virtual machines*, and what's described above a *programming language virtual machine*. These are two variations of the same idea.

A *virtual machine* is a software implementation of some real or imaginary machine that executes programs in some kind of isolation.

The Java virtual machine, for example, has an instruction set, just like a real machine. The job of the Java VM is to translate (and execute) programs written to this imaginary instruction set (Java bytecode) to programs in the instruction set of the underlying hardware (*e.g.*, x86).

An operating system VM might also be doing some translation, if the guest architecture and the host architecture are different (*e.g.*, running ARM code on *x86*), or if the underlying hardware does not provide sufficient isolation protection (as was the case with x86 chips until recently).

An operating system VM is not concerned with object allocation and collection. A programming language VM is less concerned with enforcing isolation on poorly behaved programs, because the programs it executes are known to be memory-safe.

5.3.2 *Structure of the Call Stack*

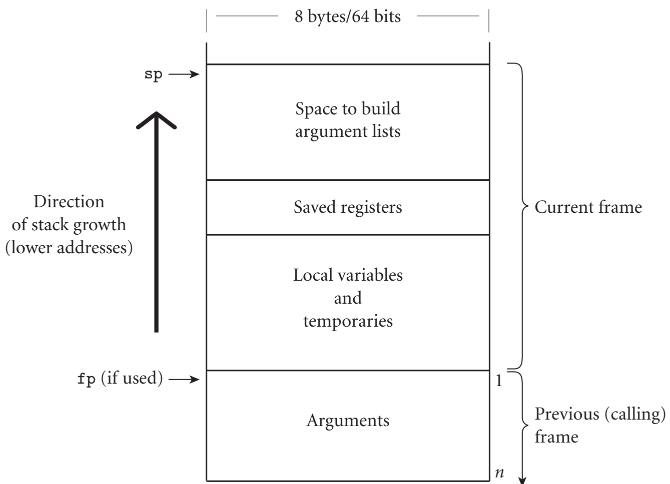
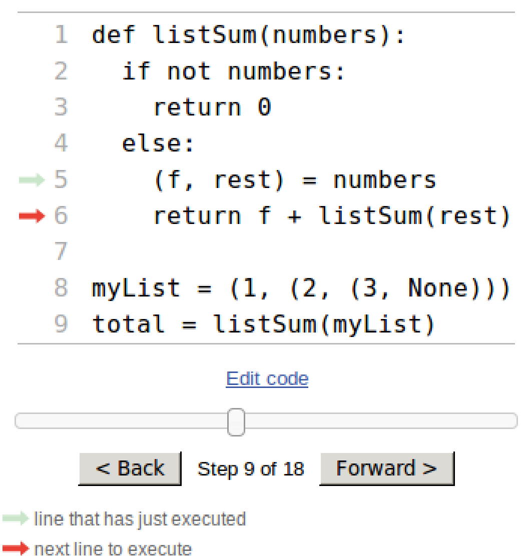


Figure 5.8: Structure of the call stack [PLP f8.11, which is essentially the same as Tiger f6.2]. By historical convention, the call stack grows from higher addresses to lower addresses. PLP draws the stack growing up (which is the normal way to visualize a stack), whereas the Tiger book draws the stack growing down (which would be the normal way to visualize growth from higher to lower addresses).

Stack frames are sometimes called *activation records*.



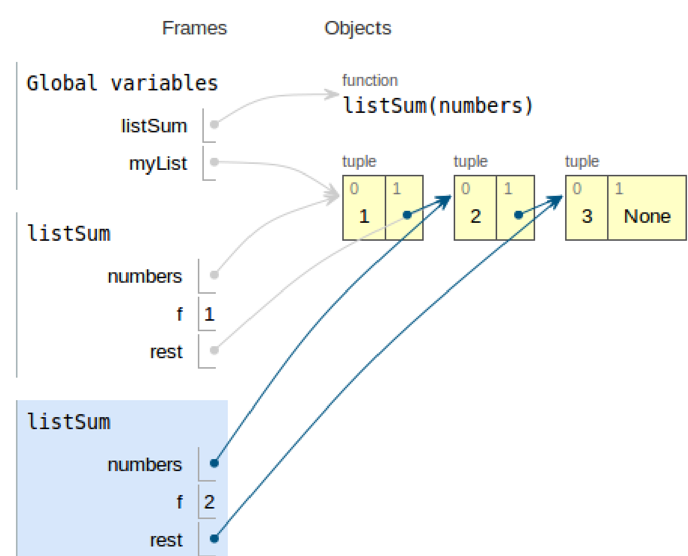


Figure 5.9: Visualization of call stack and heap from `PythonTutor.com`

5.3.3 *Object Header and Type Information Block*

Crafting: §8.5.2

Object Header

Pointer to the Type Information Block: Each object has a type, and each object header (usually) has a pointer to a record ('information block') about that type. The TIB may contain the virtual method dispatch table.

Identity HashCode: In Java (and many other languages) each object has an identity hashcode: a unique integer that is associated with it. (These are unique for all of the live objects at any given point in time; however, it is possible for a number to be reused for a different object that is created later in the execution.)

Lock: Locks are used to ensure disciplined mutation of data in a multi-threaded program.

By contrast, arrays in C (a memory unsafe language) are just chunks of memory and do not have an object header. How does one know how long the array is? By convention there is a null in the last position. The buffer overflow problem you may have heard of is about running off the end of the array and clobbering whatever happens to be in the next memory location. This can happen because of incompetence or because of malfeasance. There is no way for C to prevent this from happening because arrays do not have an object header, and so one never really knows how long the array is supposed to be.

GC bits: The garbage collector may want to associate some information with each object. A GC based on reference counting will include an integer counting the number of incoming pointers to the object. A mark-sweep GC will include some bits (usually two) to indicate that the object is still live during the mark phase (the sweep phase then collects unmarked objects).

Array length: If the object is an array, then its header will also include an integer indicating how long the array is. This is used to ensure that the array is accessed only within bounds.

Type Information Block / Class Descriptor

Super classes: Some kind of information about the super-classes to facilitate dynamic type tests (*e.g.*, instanceof checks).

VTable: The virtual method table. The addresses of where to branch to invoke polymorphic methods.

```
class foo {
    int a;
    double b;
    char c;
public:
    virtual void k( ...
    virtual int l( ...
    virtual void m();
    virtual double n( ...
    ...
} F;
```

Figure 5.10: (PLP f9.3)

```
class bar : public foo {
    int w;
public:
    void m();  //override
    virtual double s( ...
    virtual char *t( ...
    ...
} B;
```

Figure 5.11: (PLP f9.4)

```
class A           {int x = 0;
                   int f() {...} }
class B extends A {int g() {...} }
class C extends B {int g() {...} }
class D extends C {int y = 0;
                   int f() {...} }
```

Figure 5.12: (Tiger f14.3)

5.4 Cfront: Translating C++ to C

The original C++ compiler, Cfront by Bjarne Stroustrup, translated C++ to C. Let's take a look at what it did (approximately). C++ is a super-set of C: every C program is a legal C++ program. So what extra features of C++ do we need to think about? Classes, inheritance, and virtual functions (dynamic dispatch / polymorphism). What tools do we have in C to translate these things to? Structs, function pointers, arrays, switch statements, and unsafe casts.

- Each class becomes a struct, with some extra fields for the object header (just a TIB pointer in these examples).

  The struct will include all of the fields inherited from all of the super-classes (not show in these examples), as well as the fields declared locally in the class.

- A new struct called TIB is introduced for the *Type Information Block*. This will contain some information used for dynamic dispatch: possibly a *vtable* (Figures 5.13 and 5.15) or a *type tag* (Figure 5.16).

  The TIBs need to be initialized somewhere before the program starts executing. We've just put this initialization in the main method in these examples.

- Every class method becomes a standalone procedure with a new first parameter called 'this,' which is of the type of the struct that corresponds to the class in which the method was declared. Call sites to methods are modified to pass in the this parameter.

- New branching logic is introduced at polymorphic call sites. We show two possible translations here: *vtables* (Figures 5.13 and 5.15), and *switch* statements (Figure 5.16). Other translations are possible. For example, the SmallEiffel compiler generates nested ifs.

  A vtable is an array of function pointers. Each method is assigned an index into this array. The call site looks up the appropriate function pointer and branches to it.

  The switch statement translation assigns every class an integer representing its type. Polymorphic call sites then branch based on the runtime value of this integer.

The simple example program in Figure 5.13 prints a (positive) random number. The slightly more sophisticated program in Figure 5.15 prints a random number that is randomly made negative sometimes.

The examples that follow do not illustrate a number of issues, such as:

- Inheritance of fields. The field declarations are simply copied into the structs corresponding the subclasses, which is easy enough. The variables that are of the superclass type in the original source must be changed to either union types or void* in the translation. Again, that's not difficult, but it makes the examples look uglier and more complicated, because the reader would need to understand either unions or void* pointers.
- Static fields. These can be placed in the TIB.
- Other object header data, such as locks or GC bits.

C++ original

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4  using namespace std;
5
6  class RNumber {
7  public:
8      int x = rand();
9      int getX() { return x; }
10 };
11
12 int main() {
13     srand(time(NULL));
14     RNumber n;
15     int x2 = n.getX();
16     printf("x2 is %d", x2);
17     return 0;
18 }
```

C translation

```
#include <stdio.h>                                        1
#include <stdlib.h>                                       2
#include <time.h>                                         3
                                                          4
typedef struct RNumber RNumber;                           5
struct RNumber { // class becomes a struct                6
    int x;                                                7
};                                                        8
                                                          9
// We add a new parameter, "this" to procedures           10
int RNumber_getX(RNumber this) { return this.x; }         11
                                                          12
int main() {                                              13
    srand(time(NULL));                                    14
    RNumber n;                                            15
    // run the object constructor here                    16
    n.x = rand();                                         17
    // back to our regularly scheduled program            18
    int x2 = RNumber_getX(n);                             19
    printf("x2 is %d", x2);                               20
    return 0;                                             21
}                                                         22
```

Figure 5.13: Simple C++ to C translation

Figure 5.13 shows a simple translation of a C++ program to C. The class RNumber becomes a struct RNumber. The method RNumber.getX() becomes the procedure RNumber_getX(RNumber this).

The example in Figure 5.13 does not include dynamic dispatch. Therefore, we have elided the object headers and type information blocks from the C translation.

C translation (switch)

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

typedef struct RNumber RNumber;
struct TIB {
    int type;
};

// The class RNumber becomes struct RNumber
struct RNumber {
    // object header (just a TIB pointer here)
    struct TIB tib;
    // regular data members
    int x;
};

// We add a new parameter, "this"
int RNumber_getX(RNumber this) { return this.x; }

int main() {
    srand(time(NULL));
    RNumber n;
    // Set up the object metadata
    n.tib.type = 1;
    // done with object metadata
    // run the object constructor here
    n.x = rand();
    // back to our regularly scheduled program
    int x2;
    switch (n.tib.type) { // call getX()
        case 1: // RNumber
            x2 = RNumber_getX(n); break;
    } // end call getX()
    printf("x2 is %d", x2);
    return 0;
}
```

C translation (VTables)

Figure 5.14: Simple C++ to C translation with VTables and with switch statements

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

typedef struct RNumber RNumber;
struct TIB {
    int (* vtable[1]) (RNumber this);
};

// The class RNumber becomes struct RNumber
struct RNumber {
    // object header (just a TIB pointer here)
    struct TIB tib;
    // regular data members
    int x;
};

// We add a new parameter, "this"
int RNumber_getX(RNumber this) { return this.x; }

int main() {
    srand(time(NULL));
    RNumber n;
    // Set up the object metadata
    n.tib.vtable[0] = RNumber_getX;
    // done with object metadata
    // run the object constructor here
    n.x = rand();
    // back to our regularly scheduled program
    int x2 = n.tib.vtable[0](n); // call getX();
    printf("x2 is %d", x2);
    return 0;
}
```

Figure 5.15: C++ to C translation using VTables, with a polymorphic call

C++ original

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4  using namespace std;
5
6  class RNumber {
7  public:
8      int x = rand();
9      virtual int getX() = 0;
10 };
11
12 class Positive : public RNumber {
13 public:
14     virtual int getX() { return x; }
15 };
16
17 class Negative : public RNumber {
18 public:
19     virtual int getX() { return x * −1; }
20 };
21
22 int main() {
23     srand(time(NULL));
24     RNumber* poly;
25     int r = rand() % 2; // flip a coin
26     if (r) {
27         poly = new Positive;
28     } else {
29         poly = new Negative;
30     }
31     int x2 = poly−>getX(); // polymorphic call site
32     printf("x2 is %d", x2);
33     return 0;
34 }
```

C translation (VTables)

```
#include<stdio.h>                                                          1
#include<stdlib.h>                                                         2
#include<time.h>                                                           3
                                                                           4
typedef struct TIB TIB;                                                    5
typedef struct RNumber RNumber;                                            6
struct TIB {                                                               7
    int (* vtable[1]) (struct RNumber* this);                              8
};                                                                         9
                                                                           10
struct RNumber {                                                           11
    // Object header                                                       12
    TIB tib;                                                               13
    // instance fields                                                     14
    int x;                                                                 15
};                                                                         16
int Positive_getX(struct RNumber* this) { return (*this).x; }              17
int Negative_getX(struct RNumber* this) { return (*this).x * −1; }         18
                                                                           19
                                                                           20
int main() {                                                               21
    srand(time(NULL));                                                     22
    struct RNumber poly;                                                   23
    poly.x = rand(); // object constructor                                 24
    int r = rand() % 2; // flip a coin                                     25
    if (r) {                                                               26
        // Set up the object metadata (i.e., the TIBs)                     27
        poly.tib.vtable[0] = Positive_getX;                                28
    } else {                                                               29
        // Set up the object metadata (i.e., the TIBs)                     30
        poly.tib.vtable[0] = Negative_getX;                                31
    }                                                                      32
    // back to our regularly scheduled program                             33
    int x2 = poly.tib.vtable[0](&poly); // call getX();                    34
    printf("x2 is %d", x2);                                                35
    return 0;                                                              36
}                                                                          37
```

Figure 5.16: C++ to C translation using switch statements

Simple example (Figure 5.13)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4
5  typedef struct RNumber RNumber;
6  struct TIB {
7      int type;
8  };
9
10 // The class RNumber becomes struct RNumber
11 struct RNumber {
12     // object header (just a TIB pointer here)
13     struct TIB tib;
14     // regular data members
15     int x;
16 };
17
18 // We add a new parameter, "this"
19 int RNumber_getX(RNumber this) { return this.x; }
20
21 int main() {
22     srand(time(NULL));
23     RNumber n;
24     // Set up the object metadata
25     n.tib.type = 1;
26     // done with object metadata
27     // run the object constructor here
28     n.x = rand();
29     // back to our regularly scheduled program
30     int x2;
31     switch (n.tib.type) { // call getX()
32         case 1: // RNumber
33             x2 = RNumber_getX(n); break;
34     } // end call getX()
35     printf("x2 is %d", x2);
36     return 0;
37 }
```

Polymorphic example (Figure 5.15)

```
#include<stdio.h>                                                              1
#include<stdlib.h>                                                             2
#include<time.h>                                                               3
                                                                               4
typedef struct TIB TIB;                                                        5
struct TIB {                                                                   6
    int type;                                                                  7
};                                                                             8
                                                                               9
struct RNumber {                                                               10
    // Object header                                                           11
    TIB tib;                                                                   12
    // instance fields                                                         13
    int x;                                                                     14
};                                                                             15
int Positive_getX(struct RNumber* this) { return (*this).x; }                  16
int Negative_getX(struct RNumber* this) { return (*this).x * −1; }             17
                                                                               18
                                                                               19
int main() {                                                                   20
    srand(time(NULL));                                                         21
    struct RNumber poly;                                                       22
    poly.x = rand(); // object constructor                                     23
    int r = rand() % 2; // flip a coin                                         24
    if (r) {                                                                   25
        // Set up the object metadata                                          26
        poly.tib.type = 1; // Positive is type 1                               27
    } else {                                                                   28
        // Set up the object metadata                                          29
        poly.tib.type = 2; // Negative is type 2                               30
    }                                                                          31
    // back to our regularly scheduled program                                 32
    int x2;                                                                    33
    switch (poly.tib.type) { // call getX()                                    34
        case 1: // Positive                                                    35
            x2 = Positive_getX(&poly); break;                                  36
        case 2: // Negative                                                    37
            x2 = Negative_getX(&poly); break;                                  38
    } // end call getX()                                                       39
    printf("x2 is %d", x2);                                                    40
    return 0;                                                                  41
}                                                                              42
```

5.4.1 Optimizing Polymorphic Calls

Tiger: §14.7
Modern compilers are good at this.

As you can imagine, there is some performance penalty for the overhead of adding a switch statement or vtable lookup to every method call. Compiler engineers have expended a tremendous amount of effort to reduce or eliminate this overhead. For now we won't be studying those techniques, but rest assured that the overhead of using inheritance and polymorphism is often lower than the translations given in this section would suggest.

5.4.2 Why have polymorphism as a language feature?

Modularity

Adding inheritance and polymorphic calls to a language makes it more complicated for the compiler writer, but does not change the theoretical expressive power of the language: if it was Turing complete before, it's still Turing complete afterwards. So why do compiler engineers bother with the effort of supporting these language features? What do these features do for programmers?

Polymorphism makes adding new cases (classes) a modular operation. The most expensive part of the software production lifecycle is maintenance. Polymorphism, if used strategically, can reduce the cost of software maintenance by making adding new functionality to the program a more modular operation.

Consider our $\mathcal{F}$ and VHDL toolchains, for example. With the Visitor pattern (simply a strategic use of polymorphism) we can add new translations and optimizations to our compilers without having to modify our AST classes. This makes it easier to add *and test and debug* the new functionality, all of which contributes to lowering the cost of this maintenance activity.

5.4.3 Subtype Polymorphism and Parametric Polymorphism

Two different things with similar names.

We have been discussing *subtype polymorphism*, which is the most prevalent in object-oriented programming.

There is another type of polymorphism that is prevalent in functional programming and has become more common in object-oriented programming in the last ten years or so: *parametric polymorphism*. Parametric polymorphism is seen in Java's generic/parameterized types and the C++ standard template library. For example, the class `List<T>` has type parameter T, which we might instantiate with `List<String>` or `List<Integer>`, *etc.*

The Tiger book uses the term 'polymorphism' just for parametric polymorphism. The point of this subsection is just for you to be aware that there are two distinct ideas that are both described with the word 'polymorphism'.

Exercise 5.4.1 Consider the following pseudocode:

```
struct Shape { int type; int x; int y; }
String name(Shape s) {
    switch (s.type) {
        case 0: return "triangle";
        case 1: return "ellipse";
    }
}
int area(Shape s) {
    switch (s.type) {
        case 0: return x * y / 2; // triangle
        case 1: return x * y * PI; // ellipse
    }
}
```

Suppose we want to add a new kind of Shape, such as Squares. How many of these procedures would we need to edit?

Solution 5.4.1 Two: Both procedures name and area would need to be modified in order to add a new Shape such as Square.

Exercise 5.4.2 Re-write the pseudocode above using classes, inheritance, and polymorphism.

Solution 5.4.2

```
abstract class Shape {
    int x; int y;
    abstract String name();
    abstract int area();
}
class Triangle extends Shape {
    String name() { return "triangle"; }
    int area { return x * y / 2; }
}
class Ellipse extends Shape {
    String name() { return "ellipse"; }
    int area { return x * y * PI; }
}
```

Exercise 5.4.3 In this new code, what do we have to modify in order to add a new kind of Shape? (*e.g.*, Square)

Solution 5.4.3 Nothing else needs to be modified in order to add Square.

Exercise 5.4.4 Write the pseudo-code to add Square to the re-factored pseudocode above

Solution 5.4.4

```
class Square extends Shape {
    String name() { return "square"; }
    int area { return x * y; }
}
```

gcc, Eclipse, edg, gdb, ghc, HipHop, v8, XUL, CLR, WAM

Tiger: §10.0–1, §17.0, §17.2–3

Chapter 6 Optimization

The Scott book refers to 'optimization' as 'code improvement' because this is an engineering exercise rather than a mathematical one: in other words, we are trading off certain factors against other factors, not searching for the extrema of a parabola. For example, some optimizations are inverses of other optimizations. Generally speaking we are trying to save program execution time, and to do so we spend space. Sometimes things go the other way: spend time to save space. The memory hierarchy plays a role in how much space we have to spend, and how expensive (in time) that space is to use.

There are many different kinds of optimizations that compilers perform. The optimizations that we will study are based on *dataflow analysis:* an analysis of how values are computed by the program. There are a variety of optimizations, and a variety of dataflow analyses. We will look at two of each in depth.

The analyses that we consider in this chapter are *static: i.e.*, they are performed on the source code, at compile time. By contrast, a *dynamic* analysis is performed at runtime, when we have access not only to the code, but also to values for the variables. Modern virtual machines often do some dynamic analysis when optimizing programs: information from the program execution can be used to guide the optimization decisions. Dynamic analysis is an important concept to be aware of, but we will not study it in detail.

6.1 A Learning Progression Approach to Dataflow Analysis

A *learning progression* involves repetition with increasing complexity. First we see a simple variant of an idea, to grasp the general concept, then we learn how to apply it in increasingly more sophisticated settings. There are three stages in our learning progression for dataflow analysis:

a. *By Intuition.* First we'll just look at some small programs and see what needs to be done, without formalizing anything.

b. *With Equations (but no loops/branches).* Next, we'll learn about:

   - The distinction between a dataflow analysis and an optimization: an analysis tells us something about the program; an optimization transforms the program based on that information.
   - We can solve these equations by substitution (also known as Gaussian elimination). This is the technique that you learned in highschool, and is the main way that you solve systems of equations in your other engineering courses.

c. *With Equations (and loops/branches)* Now things get more complex. Once the program we are analyzing has loops or branches, then we can no longer solve the equations by substitution. We need to learn:

- Solving a system of equations by *iteration to a fixed point*. We first learned this technique in lab4.
- How to choose the initial values. This is a new challenge that we did not face in lab4. Back in lab4 we were transforming the ast, and the initial state was the input ast that the programmer had written. Now, we are just doing an analysis (we are not yet transforming the ast— that happens after the analysis is completed). This analysis involves a variety of sets. The solving works by iterating until the values of the sets reach a fixed point. To start the process, we need to pick initial values for the sets. How to pick these initial values varies depending on the equations that describe the dataflow analysis.

6.2 *Optimization by Intuition*

Crafting: §14.1.1

It is easy to do compiler optimizations by human intuition on small programs. For example:

Common Subexpression Elimination

	Example 1	*Example 2*	*Example 3*	*Example 4*

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

After	t = x + y;
	a = t + z;
	b = t * z;
		t = x + y;
		a = t + z;
		b = t * z;
		(Sort operands)
			Can't be done because the value of x changes.	t = x + y;
				a = t * z;
				b = t * z;
				Need another round of CSE to discover that t*z is common.

Loop invariant code motion: move code that doesn't change inside the loop (*i.e.*, is loop invariant) to outside of the loop.

	Example 1	*Example 2*	*Example 3*

Before	`for (i = 0; i < n; ++i) {`
	`    a[i] = x + y;`
	`}`
		`for (i = 0; i < n; ++i) {`
		`    b = x + y;`
		`    a[i] = b * b;`
		`}`
			`for (i = 0; i < n; ++i) {`
			`    b = x + y * i;`
			`    a[i] = b * b;`
			`}`

After	`t = x + y;`
	`for (i = 0; i < n; ++i) {`
	`    a[i] = t;`
	`}`
		`t = x + y;`
		`u = t * t;`
		`for (i = 0; i < n; ++i) {`
		`    a[i] = u;`
		`}`
			Can't be done because the value of b now varies with each iteration of the loop.

There are many other optimizations that can be done, such as constant propagation (which we did in lab4 as part of simplifying $\mathcal{F}$), dead code elimination, *etc.* We will focus on these two.

6.3 Optimization Step By Step

a. Convert the ast to *three address code*. Most programming languages allow the programmer to write complex expressions with multiple operators, such `a + b*c`. For analysis, we want to break those down so that each expression has only one operator, and we store the intermediate values in temporary variables. Discussed in detail below.

b. *Dataflow Analysis:* construct and solve a system of equations that describes the set of possible values the program could compute.

c. *Optimize:* transform the ast based on what we learned from the dataflow analysis.

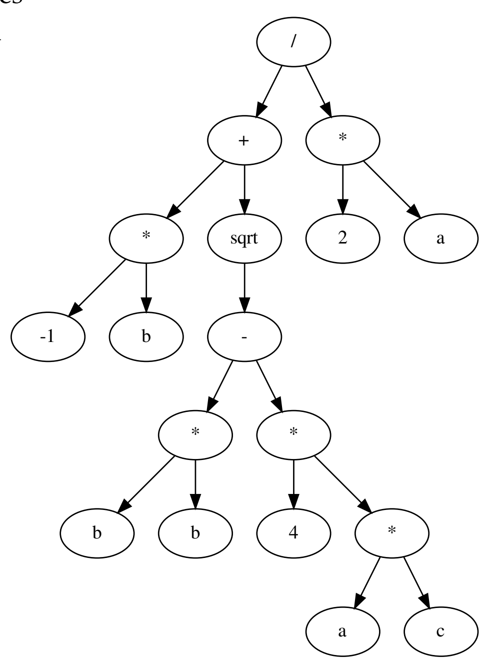
6.4 Convert to Three-Address Form

It is much easier to perform dataflow analysis and optimization on code that is in three-address form, so the first step is to convert to that form. The optimization that we intuitively above actually comprises both Common Subexpression Elimination and Copy Propagation. The Copy Propagation doesn't become clear until we convert the example to three-address form and step through it:

Copy Propagation also makes use of Reaching Definitions. It also requires some additional dataflow analyses that we will not discuss here, and so you only need to know how to do Copy Propagation by intuition, not formally.

Original	*Three-Address Form*	*After CSE*	*After Copy Prop.*	*After Dead Code Elim.*
	t = x + y;	t = x + y;	t = x + y;	t = x + y;
a = (x + y) + z;	a = t + z;	a = t + z;	a = t + z;	a = t + z;
b = (x + y) * z;	u = x + y;	u = t;	u = t;	b = t * z;
	b = u * z;	b = u * z;	b = t * z;	

Construct AST. Post-order traversal, introducing temporary variables for each node. The following example shows a complex expression of the quadratic formula in its original form, AST form, and three-address form.




```
public class Quadratic {

    public static void main(String[] args) {
        double x = positiveRoot(1, 3, −4);
        System.out.println(x);
        double x3 = positiveRoot3AC(1, 3, −4);
        System.out.println(x3);
        System.out.println(x == x3);
    }

    static double positiveRoot(int a, int b, int c) {
        return ((−1 * b) + Math.sqrt(b*b − 4*a*c)) / (2*a);
    }

    static double positiveRoot3AC(int a, int b, int c) {
        int negB = −1 * b;
        int b2 = b*b;
        int ac = a * c;
        int fourac = 4 * ac;
        int discriminant = b2 − fourac;
        double sqrt = Math.sqrt(discriminant);
        double numerator = negB + sqrt;
        int denominator = 2 * a;
        double x = numerator / denominator;
        return x;
    }
}
```

6.5 *Available Expressions Dataflow Analysis on Straightline Code*

Crafting: §14.3, §14.4

How do we formalize these optimizations so that we can program a compiler to perform them? The first step in each case is to perform a (different) *dataflow analysis*. For Common Subexpression Elimination we need an *Available Expressions* analysis, whereas for Loop Invariant Code Motion we need an *Reaching Definitions* analysis. These analyses are quite similar.

To perform Common Subexpression Elimination we need to know which expressions are available at each statement. In the above example 1, when we get to the statement assigning u we need to know that we have already computed the value of $x + y$ and stored it in the variable t: in other words, the expression $x + y$ is available at the statement assigning u. Let's number our statements so we can talk about them more concisely:

```
1  t = x + y;
2  a = t + z;
3  u = x + y;
4  b = u * z;
```

A dataflow analysis is defined by a set of equation templates involving four sets:

- *In:* set of expressions available at beginning
- *Out:* set of expression available at end
- *Gen:* set of expressions computed in block
- *Kill:* set of expressions killed by computations in block

The equation templates for Available Expressions are:

The *confluence operator* here is $\cap$. The confluence operator is the one that tells us what to do in the case when there are multiple predecessors.

$$In_s = \bigcap_{\forall\ p\ \in\ \text{predecessors}(s)} Out_p$$

$$Out_s = Gen_s \cup (In_s - Kill_s)$$

Let's instantiate these equation templates for each statement in our example:

We're going to do this on a statement basis instead of on a basic-block basis. The difference is not conceptually important.

Code to be analyzed:

```
1 t = x + y;
2 a = t + z;
3 u = x + y;
4 b = u * z;
```

$In_1 = \emptyset$	1 has no predecessors
$Gen_1 = \{$t=x+y;$\}$	the expression x+y
$Kill_1 = \{$a=t+z;$\}$	expressions that depend on t
$Out_1 = Gen_1 \cup (In_1 - Kill_1)$	instantiating the equation template
$In_2 = Out_1$	1 is the only predecessor of 2
$Gen_2 = \{$a=t+z;$\}$	the expression t+z
$Kill_2 = \emptyset$	no expressions depend on a
$Out_2 = Gen_2 \cup (In_2 - Kill_2)$	instantiating the equation template

Code to be analyzed:

```
1 t = x + y;
2 a = t + z;
3 u = x + y;
4 b = u * z;
```


$In_3 = Out_2$	2 is the only predecessor of 3
$Gen_3 = \{$u=x+y;$\}$	the expression x+y
$Kill_3 = \{$b=u*z;$\}$	expressions that depend on u
$Out_3 = Gen_3 \cup (In_3 - Kill_3)$	instantiating the equation template
$In_4 = Out_3$	3 is the only predecessor of 4
$Gen_4 = \{$b=u*z;$\}$	the expression u*z
$Kill_4 = \emptyset$	no expressions depend on b
$Out_4 = Gen_4 \cup (In_4 - Kill_4)$	instantiating the equation template

Now that we have our set of equations we can solve them. Since there are no loops or branches, we can just substitute:

Code to be analyzed:

```
1 t = x + y;
2 a = t + z;
3 u = x + y;
4 b = u * z;
```


$$Out_1 = \{\text{t=x+y;}\}$$
$$Out_2 = \{\text{t=x+y; a=t+z;}\}$$
$$Out_3 = \{\text{t=x+y; a=t+z; u=x+y;}\}$$
$$Out_4 = \{\text{t=x+y; a=t+z; u=x+y; b=u*z;}\}$$

Common Subexpression Elimination. Now that we know the Available Expressions we can do our Common Subexpression Elimination. We examine each statement and see if the expression it evaluates is already available: *i.e.*, if $In \cap Gen$ is non-empty.

We discover that statement 3 generates x+y, and that x+y is already available (*i.e.*, the expression x+y is in In_3) with the name t, so we can change statement 3 to read u=t;.

6.6 *Dataflow Analysis on Programs With Loops & Branches*

In the above examples the equations were simple enough that we could solve them symbolically, by substituting and re-arranging the equations as you have been doing since highschool. When the program we are analyzing gets more complicated — has loops or conditionals — this approach will not be enough. The loops or branches will give us statements with multiple predecessors, and that will give us equations of the following form:

$$P = P \cup Q$$

Equations like that are easy to solve when working with real numbers, as you do in most of your other engineering courses, such as:

$$X = X \times Y$$

We divide both sides by X and conclude that $Y = 1$. But in dataflow analysis, we are working with sets, not real numbers. And division is not defined for sets. So we need another way to solve.

The process of solving a system of equations by substitution is also known as *Gaussian elimination*. This is the technique that you use to solve systems of equations in most of your other engineering courses.

In algebraic terms, Gaussian elimination only works when the values of the variables are drawn from a *field*. The definition of an algebraic field requires inverse elements, *i.e.*, a concept of 'division'.

There are dataflow analysis techniques that have developed concepts of 'loop-breaking' so that Gaussian elimination can be applied. Those techniques are sometimes limited in the programs they can analyze, and are beyond the scope of this course. The iterative technique always works.

Subtraction with sets is also a bit different than with reals. When subtracting reals, the result can be less than zero. With sets, by contrast, the result of subtraction can never be less than the empty set: there is no concept of 'negative sets'. This matters because the dataflow equation templates often involve subtracting the *Kill* set from the *In* set.

6.6.1 *Iteration to a Fixed Point*

Iteration to a fixed point is a technique that we first learned in LAB4. It is the way to solve these dataflow equations. Back in LAB4, we learned that to apply this technique we need to know that the system we are working with will be *confluent:* that is, that it will *terminate* and *converge*. Those properties are still important now. Additionally, in this context we will see that picking the right initial values also matters. In LAB4 there was only one choice for initial values (the incoming AST), so we didn't have to think about it. Now we can choose to initialize our sets as either empty or full.

THERE ARE MULTIPLE FIXED POINTS for a given set of equations. For example, the following equation has potentially many solutions:

$$S = \{x\} \cup S$$

Any set that includes x is a solution to this equation, such as:

$$\{x\} = \{x\} \cup \{x\}$$
$$\{x,y\} = \{x\} \cup \{x,y\}$$
$$\{x,y,z\} = \{x\} \cup \{x,y,z\}$$

All of these solutions are *fixed points* of the equation. While they are all, mathematically, solutions, in dataflow analysis we will always want either the *greatest fixed point* (largest solution) or the *least fixed point* (smallest solution). We initialize appropriately to get the solution that we are looking for.

INITIALIZATION depends on the confluence operator used in the equations for the specific dataflow analysis. Different analyses use different confluence operators. For example, Available Expressions uses $\cap$, whereas Reaching Definitions uses $\cup$.

Confluence Operator	*Desired Solution*	*Initial In Sets*	*In Size*
$\cap$	greatest fixpoint	$\{all\}$	shrinks
$\cup$	least fixpoint	$\emptyset$	grows

THE DOMAIN of dataflow analysis sets are typically some program fragment, such as variables or expressions. Since our input program is finite, it must have a finite number of fragments. So our dataflow analysis sets must be finite. But how large? Exponential.

Suppose that our dataflow analysis is concerned with variables in the program. Suppose that there are two variables: x and y. Then our dataflow analysis sets might have any of the following possible values: $\emptyset$, $\{x\}$, $\{y\}$, $\{x, y\}$.

Let V name the set of all interesting fragments (*e.g.*, variables) in the input program, and let $n = |V|$. In the above example, $V = \{x, y\}$, and $n = 2$. Let D name the domain of possible values that the dataflow analysis sets can have. In the above example, $|D| = 2^n = 4$. We say that D is the *powerset* of V: that is, D is the set of all possible subsets of V, including the empty set and V itself.

$$D = \mathcal{P}(V) = \{\emptyset, \{x\}, \{y\}, \{x, y\}\}$$

We can visualize powersets with Hasse diagrams (Figure 6.1). These diagrams also reveal to us a structure in the powersets: some subsets include others. For example, $\{x\} \subset \{x, y\}$.

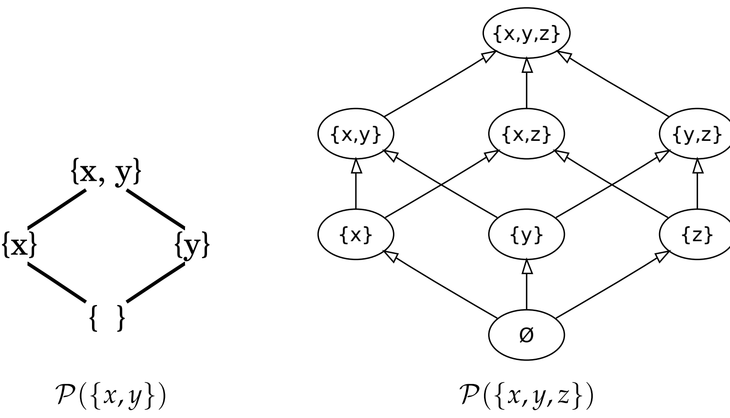


Figure 6.1: Hasse diagrams for domains of powersets of two and three elements. Images from Wikipedia, under the GNU Free Documentation License.

TERMINATION. Will dataflow analysis terminate? Let's revisit the termination argument that we saw in LAB4 for our term-rewriting system, which had three components:

Formally speaking, we say that the domain forms a *semi-lattice* with *finite chains* between top and bottom, and that the dataflow *transfer functions* are *monotone*.

- The input was finite (the original program AST).
- The transformations always made the program smaller. In other words, the transformations were *monotonic*.
- There is a bound on how small the program could get (*e.g.*, it cannot have a negative size).

So our term re-writing system in LAB4 keeps making the program smaller until it gets stuck (*i.e.*, reaches the least fixed point), and then we declare victory. We know it will always get stuck because there is a bound on how small programs can get.

We will construct a similar argument here for the termination of dataflow analyses:

- The domain of possible values for the dataflow analysis sets is finite: it is the powerset of interesting program fragments (and the original input program is finite).
- The domain has a smallest value (∅) and a largest value (the set of all interesting program fragments).
- The dataflow equations are monotonic. That is, as we go through the iterative solving technique, the dataflow equations will always push the solution in one direction in the Hasse diagram. Some dataflow analyses always push the solutions up, and some always push the solutions down.

This monotonicity property is always true, but proving it is beyond the scope of ECE351.

Similarly, a dataflow analysis keeps pushing the solutions up (or down) on the Hasse diagram until they all get stuck, and then we declare victory. In other words, the iterative solving keeps going until it reaches the greatest/least fixed point (depending on which direction it is moving).

CONFLUENCE. All dataflow equations are confluent: every computation order will produce the same fixed-point solution.

Proofs of confluence are beyond the scope of ECE351.

ITERATION ORDER. All iteration orders compute the same result (confluence), but some are faster than others. In an industrial compiler, a control-flow analysis would first be performed to determine the order in which to evaluate the sets. We will not worry about iteration order in ECE351.

6.7 *Available Expressions Dataflow Analysis (with loops)*

CONSIDER THE FOLLOWING flow graph, perform Common Subexpression Elimination by computing Available Expressions using (multi-step) iteration to the greatest fixed point.

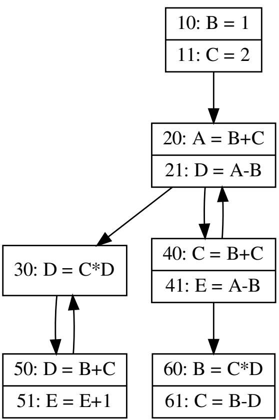
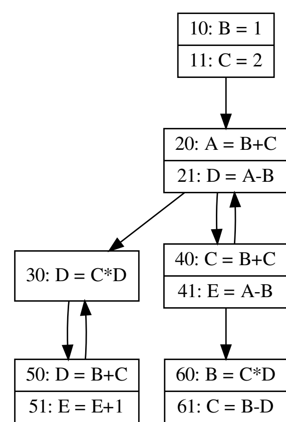


Figure 6.2: Example flow graph from *Dragon* book (1st ed.) exercise 14.1; modified by removing edge $B_3 \rightarrow B_4$. Each statement is given a line number.

By intuition we can see that lines 40 and 50 have redundant computations of B+C. We can change these lines to C=A and D=A, respectively, to eliminate the common subexpression by re-using the value of the available expression.

We can also see that line 41 has a redundant computation of A-B, and so we can change that line to E=D.

We always know the concrete solutions for the Gen and Kill sets after one step, since these sets do not depend on the ordering of the blocks (*i.e.*, do not depend on the control-flow of the program). So let's do these sets first. First the Gen sets, which are produced by examining each block in isolation:

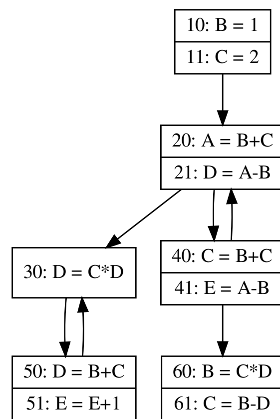


$$
\begin{aligned}
Gen_1 &= \{1, 2\} \\
Gen_2 &= \{\text{B+C}, \text{A-B}\} \\
Gen_3 &= \{\text{C*D}\} \\
Gen_4 &= \{\text{B+C}, \text{A-B}\} \\
Gen_5 &= \{\text{B+C}, \text{E+1}\} \\
Gen_6 &= \{\text{C*D}, \text{B-D}\} \\
\{all\} &= \{\text{B+C, A-B, C*D, E+1, B-D}\}
\end{aligned}
$$

Then the Kill sets. Recall that the Kill sets are produced by a complete scan of the program, ignoring the control flow (*i.e.*, ignoring the ordering of the blocks):

$Kill_1 = \{$B+C, A-B, C*D, B-D$\}$	Depends on B or C
$Kill_2 = \{$C*D, A-B, B-D$\}$	Depends on A or D
$Kill_3 = \{$C*D, B-D$\}$	Depends on D
$Kill_4 = \{$B+C, C*D, E+1$\}$	Depends on C or E
$Kill_5 = \{$C*D, B-D$\}$	Depends on D or E
$Kill_6 = \{$B+C, A-B, C*D, B-D$\}$	Depends on B or C

Now let's build the In sets symbolically. For Available Expressions, the In sets are where we encode the control-flow of the program:



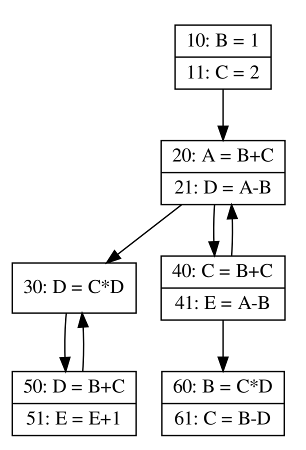
$In_1 = \emptyset$	1 has no predecessors
$In_2 = Out_1 \cap Out_4$	Blocks 1 and 4 are the predecessors of Block 2
$In_3 = Out_2 \cap Out_5$	Blocks 2 and 5 are the predecessors of Block 3
$In_4 = Out_2$	Block 2 is the only predecessor of Block 4
$In_5 = Out_3$	Block 3 is the only predecessor of Block 5
$In_6 = Out_4$	Block 4 is the only predecessor of Block 6

There is no quantifier in the *Out* equations, so they all look the same. We can commence the iterative solving. Sets are initialized to empty. Superscripts are used to indicate time points, so Out_2^3 names the value of Out_2 at step 3.

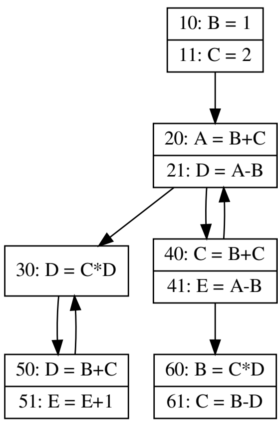
Solving for greatest fixed-point, initializing with full sets.

Set	Initial	Step 1	Step 3	Step 5
Out_1	$\Rightarrow \emptyset$	$\Rightarrow$ same	$\Rightarrow$ same	$\Rightarrow$ same
Out_2	$\Rightarrow Gen_2$	$\Rightarrow Gen_2 \cup (In_2^0 - Kill_2)$ $= \{\text{B+C, A-B}\} \cup (\{all\} - Kill_2)$ $= \{\text{B+C, A-B, E+1}\}$	$\Rightarrow Gen_2 \cup (In_2^2 - Kill_2)$ $= \{\text{B+C, A-B}\} \cup (\emptyset - Kill_2)$ $= \{\text{B+C, A-B}\}$	$\Rightarrow Gen_2 \cup (In_2^4 - Kill_2)$ = same
Out_3	$\Rightarrow Gen_3$	$\Rightarrow Gen_3 \cup (In_3^0 - Kill_3)$ $= \{\text{C*D}\} \cup (\{all\} - Kill_3)$ $= \{\text{C*D, B+C, E+1, A-B}\}$	$\Rightarrow Gen_3 \cup (In_3^2 - Kill_3)$ $= \{\text{C*D}\} \cup \{\text{B+C, A-B, E+1}\}$ $= \{\text{C*D, B+C, E+1, A-B}\}$	$\Rightarrow Gen_3 \cup (In_3^4 - Kill_3)$ $= \{\text{C*D}\} \cup \{\text{B+C, A-B}\}$ $= \{\text{C*D, B+C, A-B}\}$
Out_4	$\Rightarrow Gen_4$	$\Rightarrow Gen_4 \cup (In_4^0 - Kill_4)$ $= \{\text{B+C, A-B}\} \cup (\{all\} - Kill_4)$ $= \{\text{B+C, A-B, B-D}\}$	$\Rightarrow Gen_4 \cup (In_4^2 - Kill_4)$ $= \{\text{B+C, A-B}\} \cup \{\text{A-B}\}$ $= \{\text{B+C, A-B}\}$	$\Rightarrow Gen_4 \cup (In_4^4 - Kill_4)$ = same
Out_5	$\Rightarrow Gen_5$	$\Rightarrow Gen_5 \cup (In_5^0 - Kill_5)$ $= \{\text{B+C, E+1}\} \cup (\{all\} - Kill_5)$ $= \{\text{B+C, E+1, A-B}\}$	$\Rightarrow Gen_5 \cup (In_5^2 - Kill_5)$ $= \{\text{B+C, E+1}\} \cup \{\text{B+C, E+1, A-B}\}$ $= \{\text{B+C, E+1, A-B}\}$	$\Rightarrow Gen_5 \cup (In_5^4 - Kill_5)$ = same
Out_6	$\Rightarrow Gen_6$	$\Rightarrow Gen_6 \cup (In_6^0 - Kill_6)$ $= \{\text{C*D, B-D}\} \cup (\{all\} - Kill_6)$ $= \{\text{C*D, B-D, E+1}\}$	$\Rightarrow Gen_6 \cup (In_6^2 - Kill_6)$ $= \{\text{C*D, B-D}\} \cup \{\text{B-D}\}$ $= \{\text{C*D, B-D}\}$	$\Rightarrow Gen_6 \cup (In_6^4 - Kill_6)$ $= \{\text{C*D, B-D}\} \cup \emptyset$ $= \{\text{C*D, B-D}\}$
Set	**Initial**	**Step 2**	**Step 4**	**Step 6**
In_1	$\Rightarrow \emptyset$	$\Rightarrow$ same	$\Rightarrow$ same	$\Rightarrow$ same
In_2	$\Rightarrow \{all\}$	$\Rightarrow Out_1^1 \cap Out_4^1$ $= \emptyset \cap \{\text{B+C, A-B, B-D}\}$ $= \emptyset$	$\Rightarrow Out_1^3 \cap Out_4^3$ = same	$\Rightarrow Out_1^5 \cap Out_4^5$ = same
In_3	$\Rightarrow \{all\}$	$= Out_2^1 \cap Out_5^1$ $= \{\text{B+C, A-B, E+1}\} \cap \{\text{B+C, E+1, A-B}\}$ $= \{\text{B+C, A-B, E+1}\}$	$= Out_2^3 \cap Out_5^3$ $= \{\text{B+C, A-B}\} \cap \{\text{B+C, E+1, A-B}\}$ $= \{\text{B+C, A-B}\}$	$= Out_2^5 \cap Out_5^5$ = same
In_4	$\Rightarrow \{all\}$	$\Rightarrow Out_2^1$ $= \{\text{B+C, A-B, E+1}\}$	$\Rightarrow Out_2^3$ $= \{\text{B+C, A-B}\}$	$\Rightarrow Out_2^5$ = same
In_5	$\Rightarrow \{all\}$	$\Rightarrow Out_3^1$ $= \{\text{C*D, B+C, E+1, A-B}\}$	$\Rightarrow Out_3^3$ = same	$\Rightarrow Out_3^5$ $= \{\text{C*D, B+C, A-B}\}$
In_6	$\Rightarrow \{all\}$	$\Rightarrow Out_4^1$ $= \{\text{B+C, A-B, B-D}\}$	$\Rightarrow Out_4^3$ $= \{\text{B+C, A-B}\}$	$\Rightarrow Out_4^5$ = same

$Gen_1 = \{\text{1,2}\}$
$Gen_2 = \{\text{B+C, A-B}\}$
$Gen_3 = \{\text{C*D}\}$
$Gen_4 = \{\text{B+C, A-B}\}$
$Gen_5 = \{\text{B+C, E+1}\}$
$Gen_6 = \{\text{C*D, B-D}\}$
$\{all\} = \{\text{B+C, A-B, C*D, E+1, B-D}\}$



$Kill_1 = \{\text{B+C, A-B, C*D, B-D}\}$	Depends on B or C
$Kill_2 = \{\text{C*D, A-B, B-D}\}$	Depends on A or D
$Kill_3 = \{\text{C*D, B-D}\}$	Depends on D
$Kill_4 = \{\text{B+C, C*D, E+1}\}$	Depends on C or E
$Kill_5 = \{\text{C*D, B-D}\}$	Depends on D or E
$Kill_6 = \{\text{B+C, A-B, C*D, B-D}\}$	Depends on B or C



COMMON SUBEXPRESSION ELIMINATION: We look at the intersection of the *In* and *Gen* sets for each block, to see if a block is redundantly computing an expression that is already available.

Block	*In*	∩	*Gen*	= Result
1	∅	∩	∅	= ∅
2	∅	∩	{B+C, A-B}	= ∅
3	{B+C}	∩	{C*D}	= ∅
4	{B+C, A-B}	∩	{B+C, A-B}	= {B+C, A-B}
5	{B+C, C*D, A-B}	∩	{B+C, E+1}	= {B+C}
6	{B+C, A-B}	∩	{C*D, B-D}	= ∅

So we can change line 40 to C=A, line 50 to D=A, and line 41 to E=D.

WHAT IF WE INITIALIZED INCORRECTLY? Common subexpression elimination uses the Available Expressions dataflow analysis, which has ∩ as the confluence operator. Therefore, the *In* sets should be initialized full. What if we initialized them empty by mistake? This computation is done on the next page. Let's compare.

Block	*In (init. full)*	*In (init. empty)*	: Difference
1	∅	∅	: ∅
2	∅	∅	: ∅
3	{B+C}	{B+C}	: ∅
4	{B+C, A-B}	{B+C, A-B}	: ∅
5	{B+C, C*D, A-B}	{B+C,C*D}	: {A-B}
6	{B+C, A-B}	{B+C, A-B}	: ∅

The difference here is isolated to block five, and it so happens that it does not affect the Common Subexpression Elimination: since block five does not generate A-B, the fact that it has been excluded from *In* set does not change anything.

So the initialization mistake has not had serious consequences.

We can additionally observe that this mistake has made the resulting sets smaller: if we initialize with larger sets, then we get a larger solution; if we initialize with smaller sets, we get a smaller solution.

Is this mistake *conservative*? In other words, will such a mistake cause us to make an erroneous program transformation? This mistake, in the context of Common Subexpression Elimination, is conservative: the worst that it will do is cause us to miss an opportunity to optimize — it will not cause us to make an erroneous program transformation.

Is it possible to make an initialization mistake that could cause us to make an erroneous program transformation? Yes. In the mistake above, Available Expressions uses ∩ as the confluence operator, and so always makes the sets smaller. If we start with smaller sets, we might miss some opportunities for optimization, but we won't break the program.

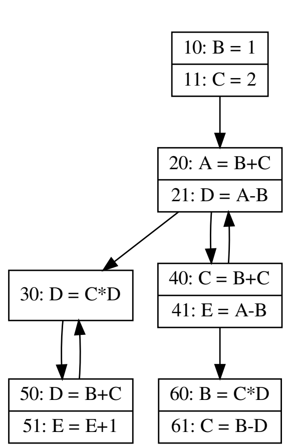
If we were performing a different dataflow analysis, such as Reaching Definitions, which has ∪ as the confluence operator, it would need the sets initialized as empty. Then it grows the sets. If, instead, we made the mistake of initializing the sets as full, then we would likely have too many values in the resulting sets (instead of too few), and that might cause us to break the program.

Solving for least fixed-point, initializing with empty sets. This is just for illustration, to show that there are multiple solutions. For available expressions we really want the greatest fixed point.

Set	Initial	Step 1	Step 3	Step 5
Out_1	$\Rightarrow \emptyset$	$\Rightarrow$ same	$\Rightarrow$ same	$\Rightarrow$ same
Out_2	$\Rightarrow \emptyset$	$\Rightarrow Gen_2 \cup (In_2^0 - Kill_2)$ $= \{$B+C, A-B$\} \cup (\emptyset - Kill_2)$ $= \{$B+C, A-B$\}$	$\Rightarrow Gen_2 \cup (In_2^2 - Kill_2)$ $=$ same	$\Rightarrow Gen_2 \cup (In_2^4 - Kill_2)$ $=$ same
Out_3	$\Rightarrow \emptyset$	$\Rightarrow Gen_3 \cup (In_3^0 - Kill_3)$ $= \{$C*D$\} \cup (\emptyset - Kill_2)$ $= \{$C*D$\}$	$\Rightarrow Gen_3 \cup (In_3^2 - Kill_3)$ $= \{$C*D$\} \cup \{$B+C$\}$ $= \{$C*D, B+C$\}$	$\Rightarrow Gen_3 \cup (In_3^4 - Kill_3)$ $=$ same
Out_4	$\Rightarrow \emptyset$	$\Rightarrow Gen_4 \cup (In_4^0 - Kill_4)$ $= \{$B+C, A-B$\} \cup (\emptyset - Kill_4)$ $= \{$B+C, A-B$\}$	$\Rightarrow Gen_4 \cup (In_4^2 - Kill_4)$ $= \{$B+C, A-B$\} \cup (\{$B+C, A-B$\} - Kill_4)$ $= \{$B+C, A-B$\}$	$\Rightarrow Gen_4 \cup (In_4^4 - Kill_4)$ $=$ same
Out_5	$\Rightarrow \emptyset$	$\Rightarrow Gen_5 \cup (In_5^0 - Kill_5)$ $= \{$B+C, E+1$\} \cup (\emptyset - Kill_5)$ $= \{$B+C, E+1$\}$	$\Rightarrow Gen_5 \cup (In_5^2 - Kill_5)$ $= \{$B+C, E+1$\} \cup (\{$C*D$\} - Kill_5)$ $= \{$B+C, E+1$\}$	$\Rightarrow Gen_5 \cup (In_5^4 - Kill_5)$ $= \{$B+C, E+1$\} \cup (\{$C*D, B+C$\} - Kill_5)$ $= \{$B+C, E+1$\}$
Out_6	$\Rightarrow \emptyset$	$\Rightarrow Gen_6 \cup (In_6^0 - Kill_6)$ $= \{$C*D, B-D$\} \cup (\emptyset - Kill_6)$ $= \{$C*D, B-D$\}$	$\Rightarrow Gen_6 \cup (In_6^2 - Kill_6)$ $= \{$C*D, B-D$\} \cup (\{$B+C, A-B$\} - Kill_6)$ $= \{$C*D, B-D$\}$	$\Rightarrow Gen_6 \cup (In_6^4 - Kill_6)$ $=$ same

Set	Initial	Step 2	Step 4	Step 6
In_1	$\Rightarrow \emptyset$	$\Rightarrow$ same	$\Rightarrow$ same	$\Rightarrow$ same
In_2	$\Rightarrow \emptyset$	$\Rightarrow Out_1^1 \cap Out_4^1$ $= \emptyset \cap \{$B+C, A-B$\}$ $= \emptyset$	$\Rightarrow Out_1^3 \cap Out_4^3$ $=$ same	$\Rightarrow Out_1^5 \cap Out_4^5$ $=$ same
In_3	$\Rightarrow \emptyset$	$= Out_2^1 \cap Out_5^1$ $= \{$B+C, A-B$\} \cap \{$B+C, E+1$\}$ $= \{$B+C$\}$	$= Out_2^3 \cap Out_5^3$ $=$ same	$= Out_2^5 \cap Out_5^5$ $=$ same
In_4	$\Rightarrow \emptyset$	$\Rightarrow Out_2^1$ $= \{$B+C, A-B$\}$	$\Rightarrow Out_2^3$ $=$ same	$\Rightarrow Out_2^5$ $=$ same
In_5	$\Rightarrow \emptyset$	$\Rightarrow Out_3^1$ $= \{$C*D$\}$	$\Rightarrow Out_3^3$ $= \{$C*D, B+C$\}$	$\Rightarrow Out_3^5$ $=$ same
In_6	$\Rightarrow \emptyset$	$\Rightarrow Out_4^1$ $= \{$B+C, A-B$\}$	$\Rightarrow Out_4^3$ $=$ same	$\Rightarrow Out_4^5$ $=$ same

$Gen_1 = \{1, 2\}$
$Gen_2 = \{$B+C, A-B$\}$
$Gen_3 = \{$C*D$\}$
$Gen_4 = \{$B+C, A-B$\}$
$Gen_5 = \{$B+C, E+1$\}$
$Gen_6 = \{$C*D, B-D$\}$
$\{all\} = \{$B+C, A-B, C*D, E+1, B-D$\}$



$Kill_1 = \{$B+C, A-B, C*D, B-D$\}$	Depends on B or C
$Kill_2 = \{$C*D, A-B, B-D$\}$	Depends on A or D
$Kill_3 = \{$C*D, B-D$\}$	Depends on D
$Kill_4 = \{$B+C, C*D, E+1$\}$	Depends on C or E
$Kill_5 = \{$C*D, B-D$\}$	Depends on D or E
$Kill_6 = \{$B+C, A-B, C*D, B-D$\}$	Depends on B or C

6.8 *Reaching Definitions Dataflow Analysis (with loops)*

To perform the *Loop Invariant Code Motion* optimization, we need to know which definitions reach where. If the definitions of all of the operands in some expression come from outside the loop then we can evaluate that expression outside the loop.

Let's consider our first example. We make variables n, x, and y into formal parameters of a procedure to give them clear definitions sites.

```
proc p(n, x, y) {
    for (i=0; i < n; i++) {
        a[i] = x + y;
    }
}
```

We convert to three-address form. In doing so, we also change the structured control flow (*i.e.*, the 'for' loop) of the source program into unstructured goto statements that a machine can actually execute. We introduce names arg1, arg2, and arg3 to talk about the binding of formal parameters x, y and z to the values they are called with.

```
1   n = arg1;
2   x = arg2;
3   y = arg3;
4   i = 0;
5   start:
6     if (!(i < n)) goto end;
7     a[i] = x + y;
8     i++;
9     goto start;
10  end:
```

The equation templates for Reaching Definitions are:

These equation templates are the same as for Available Expressions, except the intersection (∩) of the *In* equation has been changed to a union (∪). Intuitively what this says is that an expression is available if it reaches a statement on *all* incoming paths. By contrast, a definition *may* reach a statement if it comes in on *any* path.

$$In_s = \bigcup_{\forall\ p\ \in\ \text{predecessors}(s)} Out_p$$
$$Out_s = Gen_s \cup (In_s - Kill_s)$$

Note that the sets here contain definitions of variables rather than expressions: in other words, here our sets contain focus on the LHS of the statement, whereas before we were focused on the RHS.

Let's instantiate these equation templates for our example. We'll use the notation n_1 to indicate the definition of variable n on line 1.

We're going to do this on a statement basis instead of on a basic-block basis. The difference is not conceptually important.

The first four statement are trivial, so let's first state the obvious without explicitly working through all of the details:

$Out_4 = \{n_1, x_2, y_3, i_4\}$ the initial definitions reach the end of the initial block

Things get more interesting on line 6 at the top of the loop:

Code to be analyzed:

```
1  n = arg1;
2  x = arg2;
3  y = arg3;
4  i = 0;
5  start:
6    if (!(i < n)) goto end;
7    a[i] = x + y;
8    i++;
9    goto start;
10 end:
```

$In_6 = Out_4 \cup Out_8$ line 6 has two predecessors

$Gen_6 = \emptyset$ line 6 doesn't define anything

$Kill_6 = \emptyset$ line 6 doesn't define anything

$Out_6 = Gen_6 \cup (In_6 - Kill_6) = In_6$ this simplifies to In_6 since the other sets are $\emptyset$

$In_7 = Out_6$ line 7 has one predecessor

$Gen_7 = \{a_7\}$ line 7 defines a

$Kill_7 = \emptyset$ there are no other definitions of a

$Out_7 = Gen_7 \cup (In_7 - Kill_7)$ we'll solve for this later; simplifies to $Gen_7 \cup In_6$

$In_8 = Out_7$ line 8 has one predecessor

$Gen_8 = \{i_8\}$ line 8 defines i

$Kill_8 = \{i_4\}$ the definition of i on line 4 is no longer valid

$Out_8 = Gen_8 \cup (In_8 - Kill_8)$ we'll solve for this later

$In_{10} = Out_6$ line 10 follows from the goto on line 6

$Gen_{10} = \emptyset$ line 10 doesn't define anything

$Kill_{10} = \emptyset$ line 10 doesn't define anything

$Out_{10} = Gen_{10} \cup (In_{10} - Kill_{10})$ note: $Gen_{10} = Out_6 = In_6$

Solve for the interesting sets. Remember that unknown values, such as Out_8, are initialized to $\emptyset$). Note that we can use values computed in the same step: *e.g.*, when we compute Out_7 in step 1, we can use the value of In_6 that was computed previously in step 1.

This is why the iteration order of our solving matters in practice. While theory tells us that all iteration orders produce the same end result, some of them get there faster.

Simplified Formula	**Step 1**	**Step 2**
$In_6 = Out_4 \cup Out_8$	$= \{n_1, x_2, y_3, i_4\}$	$= \{n_1, x_2, y_3, i_4, a_7, i_8\}$
$Out_7 = Gen_7 \cup In_6$	$= \{n_1, x_2, y_3, i_4, a_7\}$	$= \{n_1, x_2, y_3, i_4, a_7, i_8\}$
$Out_8 = Gen_8 \cup (Out_7 - Kill_8)$	$= \{n_1, x_2, y_3, a_7, i_8\}$	$= \{n_1, x_2, y_3, a_7, i_8\}$

Code to be analyzed:

```
1  n = arg1;
2  x = arg2;
3  y = arg3;
4  i = 0;
5  start:
6    if (!(i < n)) goto end;
7    a[i] = x + y;
8    i++;
9    goto start;
10 end:
```

To perform the Loop Invariant Code Motion optimization, we need to know that In_7 does not contain any definitions of x or y from inside the loop. We see above that $In_7 = Out_6 = In_6 = \{n_1, x_2, y_3, i_4, a_7, i_8\}$. The only definitions of x and y there come from outside the loop, so we can move the computation of x+y before the loop.

In other words, both x and y are *loop invariant:* they do not change with each iteration of the loop.

6.9 *Duality of Available Expressions and Reaching Definitions*

	Available Expressions	*Reaching Definitions*
Focus	RHS (exprs)	LHS (vars)
Confluence Operator	intersection ($\cap$)	union ($\cup$)
Paths	all	any
Necessity	must	may
Initialize *In* sets	$\{all\}$	$\emptyset$
Fixed point	greatest	least

Dataflow analyses are typically parameterized by their equation templates (including the confluence operator), and how the *Gen* and *Kill* sets are computed. Because of this regular structure it is possible to build dataflow analysis frameworks.

```
1  a = t + z;
2  if (c) {
3      t = x;
4  }
5  b = t + z;
```

Figure 6.3: An example illustrating the difference between Available Expressions and Reaching Definitions. The expression t+z from line 1 is *not* available on line 5 because the value of t may have changed: it may not be the same on *all* paths leading to line 5.

The definitions of variable a on line 1 and variable t on line 3 both reach line 5. Reaching Definitions asks whether a definition *might* reach a statement *any* path. Available expressions asks whether an expression *must* reach a statement on *all* incoming paths.

6.10 Summary

- Be able to perform the following optimizations by intuition:
  - Common Subexpression Elimination
  - Copy Propagation
  - Dead Code Elimination
  - Loop Invariant Code Motion
  - Live Variables
- Be able to perform the following optimizations by dataflow analysis:
  - Common Subexpression Elimination (Available Expressions)
  - Loop Invariant Code Motion (Reaching Definitions)
- Basic skills:
  - Convert to three-address form
  - Solve for least fixed-point
  - Draw a Hasse diagram for a powerset
- *Function inlining:* We saw a version of this in the labs, in VHDL terms 'elaboration'.

Thought questions

a. What order to apply optimizations in?

b. How many times to apply them?

c. Do we know that this process will converge?

d. Will it really make the program faster/smaller/*etc.*?

e. Will we reach an optimum?

Chapter 7
Storage Management

7.1 Register Allocation

Tiger: §11.0, 21.0

Crafting: §13.3

To see a World in a Grain of Sand
And a Heaven in a Wild Flower
Hold Infinity in the palm of your hand
And Eternity in an hour
— *Auguries of Innocence*
William Blake, 1803

What you need to be able to do on the exam is draw an interference graph, colour it, and report how many registers are needed.

The software illusion: unbounded space in a finite machine.

$$\frac{\text{Turing Machine}}{\text{infinite tape}} \approx \frac{\text{Real Computer}}{\text{registers + cache + RAM + disk}}$$

We see this illusion at play all over computer engineering:

- *Operating Systems:* When we run out of RAM then we *swap* to disk (virtual memory also sometimes has hardware support, so chip designers might be involved).
- *Databases:* One of the main sources of complexity in a database engine is managing data that are too big to fit in RAM. It is much easier to write an in-memory database engine.
- *Chips:* The chip designer determines the policy of what data are duplicated in the cache, and what are only in main memory.
- *Compilers:* The compiler engineer determines how registers are *allocated* to the local variables in the program, and which are *spilled* to main memory. The programmer has the illusion of an unbounded number of registers (local variables).

  Some programming languages, such as Scheme, provide the programmer with integers of unbounded bitwidth, so the programmer never suffers from overflow. For efficient execution, the compiler engineer wants to map Scheme integers to machine integers where possible, and then transparently switch representations when the machine integer is about to overflow.

We are focused on register allocation here. The point is that the illusion of the infinite in the finite also exists elsewhere in programming languages.

Register allocation is an NP-complete problem. Intuitively we can see that it is like bin-packing: we have a bunch of local variables that we have to pack into the available registers. The main approach for register allocation is *graph colouring*. Graph colouring is another

The Tiger book's chapter on register allocation is largely devoted to explaining the details of some polynomial approximations. We're not concerned with that. You can learn it from a book later if necessary. All you need to learn now is how to model the register allocation problem as a graph colouring problem.
In class we did the example problem on page 221 (Graph 11.1).
Aside: By the *four colour theorem* we know that any planar graph requires at most four colours. The history of this theorem and its proof is an important and interesting story in the interplay between computers and mathematics. `http://en.wikipedia.org/wiki/Four_color_theorem`

well-known NP-complete problem for which there exist known polynomial approximations that perform reasonably well.

VARIABLE LIVENESS appears, on the surface, slightly different in three-Address code *vs.* assembly. The 3AC sometimes looks like it needs more registers than it does. Consider the 3AC statement f := g * h. How many registers does this require? 3? or 2? If we look at it in assembly it becomes clearer:

Instruction	*Begin # Reg.*	*End # Reg.*	*Comment*
load g	2: g, h	1: h	Load g on to the stack
load h	1: h	0	Load h on to the stack
mult	0	0	result on top of stack
store f	0	1: f	store result in f (pop off stack)

For some machines this might just be one instruction like mult $4, $5, $4, saying multiply the values in register numbers 4 and 5 and store the result in register 4: only two registers are required.

The Wikipedia entry on Register Allocation covers much of what we discussed in class. The relevant parts of that entry are excerpted here for your convenience.

Wikipedia entries are made available under a Creative Commons Attribution-ShareAlike 3.0 licence, which grants the reader the freedom to adapt the entry as long as (1) Wikipedia is acknowledged as the source of the material, and (2) the adapted material is made available to others under similar licence terms.

WIKIPEDIA EXCERPT. In compiler optimization, register allocation is the process of assigning a large number of target program variables onto a small number of CPU registers. Register allocation can happen over a basic block (local register allocation), over a whole function/procedure (global register allocation), or in-between functions as a calling convention (interprocedural register allocation).

In many programming languages, the programmer has the illusion of allocating arbitrarily many variables. However, during compilation, the compiler must decide how to allocate these variables to a small, finite set of registers. Not all variables are in use (or 'live') at the same time, so some registers may be assigned to more than one variable. However, two variables in use at the same time cannot be assigned to the same register without corrupting its value. Variables which cannot be assigned to some register must be kept in RAM and loaded in/out for every read/write, a process called spilling. Accessing RAM is significantly slower than accessing registers and slows down the execution speed of the compiled program, so an optimizing compiler aims to assign as many variables to registers as possible. Register *pressure* is the term used when there are fewer hardware registers available than would have been optimal; higher pressure usually means that more spills and reloads are needed.

Through liveness analysis, compilers can determine which sets of variables are live at the same time, as well as variables which are involved in move instructions. Using this information, the compiler can construct a graph such that every vertex represents a unique variable in the program. Interference edges connect pairs of vertices which are live at the same time, and preference edges connect pairs of vertices

We will not worry about preference edges. We will focus on interference edges.

which are involved in move instructions. Register allocation can then be reduced to the problem of K-coloring the resulting graph, where K is the number of registers available on the target architecture. No two vertices sharing an interference edge may be assigned the same color, and vertices sharing a preference edge should be assigned the same color if possible. Some of the vertices may be precolored to begin with, representing variables which must be kept in certain registers due to calling conventions or communication between modules. As graph coloring in general is NP-complete, so is register allocation. However, good algorithms exist which balance performance with quality of compiled code.

We will not worry about precolouring.

The Tiger Book discusses some of these polynomial approximations. We aren't concerned with them. Once you understand the concepts you can learn those approximations from the book later if your job requires it.

7.1.1 *Liveness Analysis*

Liveness analysis tells us, for each statement, which variables are currently holding values that might be used in the future (*i.e.*, which variables are *live*). Liveness analysis is the first step in register allocation. A statement *uses* a variable that appears on the RHS of the statement, and *defines* a variable that appears on the LHS. The dataflow equation templates for liveness analysis are:

$$In_s = Use_s \cup (Out_s - Def_s)$$

$$Out_s = \bigcup_{\forall\, x\, \in\, \text{successors}(s)} In_x$$

Consider, for example, the following block of code, accompanied by the columns on the right that indicate when each variable is live.

		b	c	d	e	f	g	h	j	k	m
live-in	k, j								|	|	
g	:= mem[j+12]						\		|	|	
h	:= k - 1						|	\	|	/	
f	:= g * h					\	/	/	|		
e	:= mem[j + 8]				\	|			|		
m	:= mem[j+16]				|	|			/		\
b	:= mem[f]	\			|	/					|
c	:= e + 8	|	\		/						|
d	:= c	|	/	\							|
k	:= m + 4	|		|						\	/
j	:= b	/		|					\	|	
live-out	d, k, j			|					|	|	

Figure 7.1: Liveness analysis example. The code block starts with the statement assigning g and ends with the statement assigning j. We are told that variables k and j are live coming into the block; variables d, k, and j are live going out of the block. The columns on the right show the results of our liveness anaylsis by intuition. A backslash (\) indicates the statement on which a variable is defined. A bar (|) indicates a statement on which a variable is live. A forward slash (/) indicates a statement on which a definition is last used. The reason for this notation is to show, for example, that in the statement $k = m + 4$ that k and m do not interfere with each other (*i.e.*, they can share the same register).
[Adapted from *Tiger* §11.1]

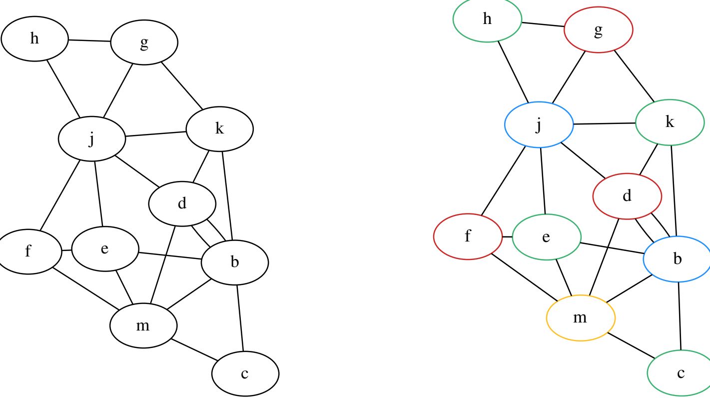


Figure 7.2: Intereference graph example corresponding to liveness analysis in Figure 7.1, before and after colouring. This colouring shows that we need four registers. The general problem of graph colouring is NP-complete. If our machine has fewer registers than are needed to colour the graph, then some colours must be *spilled* to main memory.

7.1.2 *Interference Graph Colouring*

From the liveness analysis we construct an *intereference graph*, with a node for each variable, and an edge between variables that are live at the same time. We then colour this graph, with each colour representing a register.

7.2 Garbage Collection

Tiger: §13.0–4, §13.7.1

We reviewed the basic ideas of *reference counting*, *mark and sweep*, *copying collection*, and *generational collection* using the examples from the Tiger book.

Generational collection enables faster object allocation (which is well described in the Tiger book).

```
/** It's awful: nothing happens twice. */
public class Beckett {
    public static void main() {
        act1();
        System.out.println("Waiting");
        act2();
        System.out.println("Godot");
    }
    static void act1() {
        Person vladimir = new Person();
        Person estragon = new Person();
        vladimir.friend = estragon;
        estragon.friend = vladimir;
    }
    static void act2() {
        Person lucky = new Person();
        Person pozzo = new Person();
        pozzo.friend = lucky;
        // what is the existential status
        // of Vladimir and Estragon at
        // the end of act 2?
        System.out.println("for");
    }
}

class Person {
    Person friend;
    Person() {
        super();
        friend = null;
    }
}
```


Figure 7.3: Garbage collection in action. First we draw the stack and the heap, as we have done on the board many times throughout the term, and as is done by `PythonTutor.com` and Jeliot (for Java programs).

There is no frame for act1() on the stack, because act1() is not currently executing (the point of execution pictured is at the existential status comment).

The local variables lucky and pozzo refer to the objects Person3 and Person4, respectively. The objects Person3 and Person4 have no knowledge that they are pointed to by local variables named lucky and pozzo. We see that Person4 (pozzo) has a friend, but that Person3 (lucky) does not.

The objects Person1 and Person2 were formerly known by the local variables vladimir and estragon, respectively, when act1() was executing (*i.e.*, had an active frame on the stack).

Every garbage collection strategy except for reference counting starts from the pointers on the stack and traverses the heap to determine which objects are reachable (*i.e.*, live). In this case we see that Person3 and Person4 are reachable, whereas Person1 and Person2 are not reachable. The unreachable objects are garbage and can be collected.

Reference counting, by contrast, looks at each object and its count of incoming pointers. Person1 and Person2 both have one incoming pointer, so reference counting would not identify them as garbage. Cyclic structures such as this are the main weakness of reference counting. Reference counting is the easiest garbage collection strategy to implement, but is the worst performing.

An old MIT AI Lab Koan: One day a student came to David Moon and said: 'I understand how to make a better garbage collector. We must keep a reference count of the pointers to each cons (object).' Moon patiently told the student the following story: 'One day a student came to Moon and said: "I understand how to make a better garbage collector...

7.3 *Three options for cleanup*

Once we start using the heap then we have to start worrying about cleaning up after ourselves — *garbage collection*. Turing Machines have infinite storage, but our real computers do not. When our program no longer needs some object then we want to reclaim that space to we can reuse it for some future object. There are three main strategies for handling garbage collection:

- *Don't.* The program in Figure **??** does not really need any storage reclamation, because all of the objects allocated are needed until the program terminates, at which point the entire heap is garbage. Some command-line programs in the real world are like this. Interactive programs are not like this: they do some work, clean up after themselves, then do some more work. Interactive programs include any GUI (graphical user interface) or web server.
- *Manual.* Some programming languages, such as C/C++, allow the programmer to manually reclaim memory (the `delete` keyword in C++). For the example program in Figure **??** we could delete the entire search tree after the search is performed. Deleting the entire tree would be a bit tricky:
  - We would need to implement a destructor for both the `Tree` class and the `Node` class, which would have to perform a post-order traversal of the tree (*i.e.*, the tree needs to be deleted from the leaves up).
  - We would need to be confident that the tree was well-formed. Otherwise we might fail to delete some nodes.
  - We would need to be confident that the nodes in the tree were not shared with any other trees.
  - We would need to develop, and document, a policy about whether deleting the tree would also delete the contents of the tree ('rock', 'paper', 'scissors', 'prize' in our example). In our example program in Figure Figure **??** the input tokens are shared between the tree, the `args` array, and some local variables (*e.g.*, `toFind`). So it might be unwise to delete them when we delete the tree. But it also might be tricky to remember to delete them later, because that would require knowing exactly how those objects were shared and that any incoming pointers were no longer needed.
- *Automatic.* We can rely on an *automatic garbage collector* to reclaim space used by objects that are no longer needed. Most modern programming languages include a garbage collector. We will study different techniques for automatic garbage collection later.

7.4 *Reference Counting*

Recall from §0.3 that all of our design choices involve engineering trade-offs between different available resources.

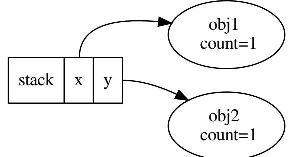
Reference Counting is the easiest technique to implement, but performs poorly in both space and time: it requires an integer for each object to record the reference count; it has difficulty with cycles; and it takes time at each *assignment* (not each allocation). Suppose the user writes an assignment statement such as the following:

```
x = y;
```

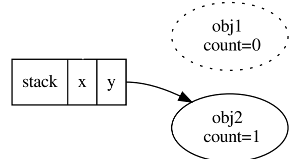
The compiler then needs to insert code such as the following:

```
// compiler generated code before user's assignment
// decrement the reference count of whatever object is currently
// referred to by variable x (should check for null first)
x.refcount−−;
// collect the object referred to by x if its counter has gone
// down to zero
if (x.refcount == 0) { delete x; }

// user's assignment
x = y;

// compiler generated code after user's assignment
// increment reference counter of object referred to by x
// (which is now referred to by both x and y)
x.refcount++;
```

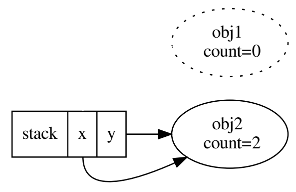
Beginning heap:



After deleting obj1 because it's count went to zero:



After assignment and count update completed:



7.5 *Mark & Sweep*

See the Tiger §13 excerpt in the ECE351 notes repo.

- Start at *root set:* local + global variables.
- Mark all objects that are reachable from the root set.
- Sweep phase: add all non-reachable objects to free-list.
- Requires slow allocation (§7.9) because heap is never defragmented.

7.6 *Semi-Space Copying Collection*

See the Tiger §13 excerpt in the ECE351 notes repo.

- Divide the heap in two halves: green and purple.
- First allocate in the green half (using fast allocation §7.10)
- When green half fills up, copy live objects over to purple half.
- Now allocate in purple half (fast allocation)
- When purple half fills up, copy live objects back to green half.
- Repeat.
- Note: when copying, *defragment* the heap.

7.7 *Generational Collection*

See the Tiger §13 excerpt in the ECE351 notes repo.

- Could be considered an assymetrical kind of copying collection.
- New objects always allocated in *nursery* (fast allocation §7.10)
- When nursery fills up, copy live objects to *old space*.
  Nursery collectsions are relatively frequent.
- When old space fills up, collect it with Mark+Sweep. Infrequent.
- Keep empirical observation: most objects (≥90%) are short-lived.
  Therefore, few objects are ever promoted to old space.

7.8 *Discussion*

- There are always engineering trade-offs.
- Most modern industrial VMs use generational collection, because it has the best performance (most of the time) and because the engineering effort to implement it has economic justification.
- Reference counting with (slow) free-list based allocation is the easiest to implement, but performs the worst.
- Overall performance of a memory management system depends on both allocation and collection costs: these cannot be considered in isolation.
- Different programs, and even different inputs to the same program, might exercise the memory management system in different ways, so there is no one strategy that is the best in all circumstances. Engineers need to do experiments to determine which strategies perform the best in the most common circumstances.

For example, there are many configuration options for the user to tune the Oracle Java VM's memory management strategies. See, for example, `http://www.cubrid.org/blog/dev-platform/how-to-tune-java-garbage-collection/` (That is not required reading.)

7.9 *Object Allocation with Free Lists*

See the Tiger §13 excerpt in the ECE351 notes repo.

- Organize all free chunks of memory into a linked list (the *free-list*).
- When the program requests space for an object in the heap, do linear search on the free-list until space of suitable size is found.
- When an object is collected, add its space to the free-list.
- Slow. Easy to implement.
- There are more sophisticated ways to manage the search rather than a free-list. But some kind of search is necessary if the heap is not periodically de-fragmented, and search will always be slower than a bump allocator (§7.10).

7.10 *Fast Object Allocation*

See the Tiger §13 excerpt in the ECE351 notes repo.

Any kind of copying collector enables faster object allocation by defragmenting the heap. This kind of allocator is sometimes called a *bump* allocator, because it just increments (bumps) a pointer:

```
void *malloc(int n) {
    if (heapTop − heapStart < n)
        doGarbageCollection();

    void *wasStart = heapStart;
    heapStart += n;
    return wasStart;
}
```

http://www.ibm.com/developerworks/library/j-jtp09275/

7.11 DieHard: Probabilistic Memory Safety for C

A *memory-unsafe* language is one in which programs can read and write to arbitrary memory locations, like C. A *memory-safe* language is one in which programs can read and write only objects, and only in pre-defined ways (*e.g.*, reading or writing to a field).

Memory-unsafe languages are required for certain operating system and device driver kinds of programs: programs whose entire point is to manipulate a machine.

Memory-safe languages, by contrast, are arguably better for programs whose purpose is to manipulate symbolic or numerical data (*i.e.*, most application programs). Memory-safe languages are better for these tasks because they prevent entire classes of bugs that can occur in memory-unsafe languages: buffer overflows, dangling pointers, double deletes, reads of uninitialized data, *etc.*.

Suppose we have a program written in C. This program has some memory errors. We don't want to debug it, and we don't want to rewrite it in a memory-safe language (which would remove these errors). What can we do? DieHard.

DieHard: Probabilistic Memory Safety for Unsafe Languages

Emery D. Berger
Dept. of Computer Science
University of Massachusetts Amherst
Amherst, MA 01003
emery@cs.umass.edu

Benjamin G. Zorn
Microsoft Research
One Microsoft Way
Redmond, WA 98052
zorn@microsoft.com

Abstract

Applications written in unsafe languages like C and C++ are vulnerable to memory errors such as buffer overflows, dangling pointers, and reads of uninitialized data. Such errors can lead to program crashes, security vulnerabilities, and unpredictable behavior. We present DieHard, a runtime system that tolerates these errors while probabilistically maintaining soundness. DieHard uses randomization and replication to achieve *probabilistic memory safety* by approximating an infinite-sized heap. DieHard's memory manager randomizes the location of objects in a heap that is at least twice as large as required. This algorithm prevents heap corruption and provides a probabilistic guarantee of avoiding memory errors. For additional safety, DieHard can operate in a replicated mode where multiple replicas of the same application are run simultaneously. By initializing each replica with a different random seed and requiring agreement on output, the replicated version of DieHard increases the likelihood of correct execution because errors are unlikely to have the same effect across all replicas. We present analytical and experimental results that show DieHard's resilience to a wide range of memory errors, including a heap-based buffer overflow in an actual application.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Dynamic storage management; D.2.0 [*Software Engineering*]: Protection mechanisms; G.3 [*Probability and Statistics*]: Probabilistic algorithms

General Terms Algorithms, Languages, Reliability

Dangling pointers: If the program mistakenly frees a live object, the allocator may overwrite its contents with a new object or heap metadata.

Buffer overflows: Out-of-bound writes can corrupt the contents of live objects on the heap.

Heap metadata overwrites: If heap metadata is stored near heap objects, an out-of-bound write can corrupt it.

Uninitialized reads: Reading values from newly-allocated or unallocated memory leads to undefined behavior.

Invalid frees: Passing illegal addresses to `free` can corrupt the heap or lead to undefined behavior.

Double frees: Repeated calls to `free` of objects that have already been freed cause freelist-based allocators to fail.

Tools like Purify [18] and Valgrind [28, 35] allow programmers to pinpoint the exact location of these memory errors (at the cost of a 2-25X performance penalty), but only reveal those bugs found during testing. Deployed programs thus remain vulnerable to crashes or attack. Conservative garbage collectors can, at the cost of increased runtime and additional memory [12, 20], disable calls to `free` and eliminate three of the above errors (invalid frees, double frees, and dangling pointers). Assuming source code is available, a programmer can also compile the code with a safe C compiler that inserts dynamic checks for the remaining errors, further increasing running time [1, 3, 27, 41, 42]. As soon as an error is detected, the inserted code aborts the program.

7.12 Memory Safety and Language Selection

Crafting: §1.6
definitions

We say that a programming language is *memory safe* if it permits reads and writes of memory only where objects have been defined. We say that a programming language is *memory unsafe* if it permits reads and writes of arbitrary locations of memory. For example, in C we can write a program that asks the user for an integer, and then writes data to that location in memory.

example safe and unsafe languages

The most common unsafe languages are C and C++. Almost all other popular languages are memory safe: Java, C#, Python, Javascript, Ruby, Perl, Haskell, ML, Basic, Pascal, VisualBasic, Lisp, *etc.* Having said that, almost all programs written in C/C++ are intended to behave in a memory safe manner: that is, the programs are not expected to write past the end of their arrays (buffer overflow), or read uninitialized memory, *etc.* There are a wide variety of tools for trying to catch these memory errors in C/C++ programs.

safety vs. performance

Memory safety usually incurs some runtime overhead to ensure that array operations are within bounds, that casts are legal, *etc.*— and, of course, automatic garbage collection. The perceived runtime cost of memory safety is often the crux of contention in language selection.

reasons for choosing a language

There are five main reasons usually given for choosing a particular language for a project:

- what kinds of errors the language admits or prevents;
- the imagined runtime performance;
- the convenience of expressing the required computation;
- the computing environment (including available libraries);
- and the skills of the available programmers.

The last two factors are extrinsic to the language and so are not part of our discussion here. On another day in this course we might be concerned with the convenience of expression. Our focus here is on the first two criteria: errors and performance.

systems programs vs. application programs

There are very few programs that need to perform memory unsafe operations: operating system kernels, device drivers, and garbage collectors are three main kinds of what we might call *systems* programs. The vast majority of programs are *applications* that are expected to behave in a memory safe manner. If a program is intended to behave in a memory-safe manner then, all other things being equal, it is better to write it in a memory-safe language in order to guarantee that the program does not have memory errors.

use a memory safe language for applications

When Java first came out it was 10–100x slower than C. That is no longer true.

Sometimes the imagined runtime cost of memory safety is cited as a reason to use a memory unsafe language for application programming. I will give the following arguments against that position:

5 reasons why to use memory safe languages for application programming

- *Practical Economics:* The major costs of producing software are testing and maintenance. For example, Microsoft hires testers and developers in equal measure. Maintenance is the most expensive part of the software life cycle. Writing in a memory safe language reduces these costs because it eliminates entire classes of bugs. In Tony Hoare's 1980 Turing Award acceptance speech[1] he talked about how he had written a compiler in the 1960's that inserted dynamic array bounds checks, and how even at that time his customers felt the reduction in bugs was well worth the runtime cost.

[1] C. A. R. Hoare. The emperor's old clothes. *Communications of the ACM*, 24 (2):75–83, February 1981. Acceptance speech for 1980 Turing Award

- *Theoretical Economics:* Greenspun's Tenth Rule states that 'Any sufficiently complicated C or Fortran program contains an ad hoc, informally-specified, bug-ridden, slow implementation of half of Common Lisp.' For example, one of the students in the s2013 offering of this course who has worked on Microsoft PowerPoint reports that PowerPoint contains its own smart-pointer implementation. *Division of labour* has, arguably, been the basis of economic advancement in the west for the last 250 years or so. On this general economic theory, compiler engineers should work hard to make memory safety less expensive, and application programmers should use memory safe languages. Having application programmers attempt to build memory safety onto an unsafe language for each project is economically inefficient.

`http://en.wikipedia.org/wiki/Greenspun's_tenth_rule`

- *Performance will be parallel*. Several years ago we hit the upper bound of single-core performance. Performance gains of the future will come from multi-core parallelism, just as the main performance gains of the 90's came from instruction-level parallelism within a single core. Application programmers should focus their efforts on high-level parallelization (and use languages and libraries that reduce the probability of parallelization errors).
- *Memory hierarchy is the bottleneck.* In modern computers the bottleneck is the memory hierarchy: the cpu sits around idle waiting for data to come in from main memory. The runtime overhead of memory safety typically does not antagonize this situation: *i.e.*, we can have it for free. For example, the Varnish http accelerator program used by Facebook and other large websites makes extensive use of assertions: approximately 10% of the non-comment source lines are protected by an assertion. Varnish's author Poul-Henning Kamp (a longtime FreeBSD contributor) recently argued[2] that these runtime checks incur almost no runtime performance

[2] Poul-Henning Kamp. My compiler does not understand me. *Communications of the ACM*, 55(7):51–53, 2012

penalty because *(a)* modern compilers are good at statically removing these runtime checks, and *(b)* the cpu is sitting idle much of the time waiting for data from main memory, so it might as well perform some safety checks on the data it already has in registers.

- *Performance is empirical.* Until you have a working system to profile, you don't know where the real performance problems are. The assumption that the real problems are caused by the overhead of language features is likely wrong. Disk i/o, network latency, and algorithmic deficiencies all cause much larger overhead.

  One possible approach is to write a prototype in a memory safe language to think through all of the conceptual issues, and then write a production version in a memory unsafe language. Gerwin Klein's team did this for their l4 microkernel operating system and found that it reduced their development costs as compared to other teams developing l4 microkernels.[3]

Take ece459 *Programming for Performance* if you really care about performance.

[3] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, USA, October 2009

Is there ever a time when performance concerns make it worthwhile to forego memory safety? Maybe, in the following cases:

sat *is an exception*

- The program does not require dynamic memory allocation (*i.e.*, all allocation is done on the stack). A more restrictive form of this rule is that the program does not require objects, just primitive int and float values, and perhaps arrays.
- The program is batch-mode (*i.e.*, not interactive) and short-running. If the program simply reads an input, computes an output, and terminates, then garbage collection might not be necessary. Any interactive program, whether via a gui or via the web, will likely require dynamic memory allocation and garbage collection.
- The program implements a single, well-understood algorithm.
- The program does not accept inputs from unknown adversaries (*i.e.*, does not accept inputs on the web). Lack of memory safety creates all kinds of security vulnerabilities, such as buffer overflow, so if the program is running in a dangerous environment then it should be written in a memory safe language.

A sat solver is an example of a non-systems program that can reasonably be written in a memory unsafe language: it implements a well-known algorithm[4]; it operates in batch mode; it does not accept inputs from adversarial users; it has restricted memory usage; it is typically small enough to be well understood by a single person (less than 5,000 lines of code); and there are well-defined ways to measure performance (an annual international competition with multiple categories). Data compression programs are a similar example.

[4] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960; and Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, 1962

Appendix B
Bibliography

[1] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft. In Eerke A. Boiten, John Derrick, and Graeme Smith, editors, *Proceedings of the 4th Integrated Formal Methods (IFM) Conference*, pages 1–20, Canterbury, UK, April 2004.

[2] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.

[3] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, 1962.

[4] Edsgar W. Dijkstra. The humble programmer. *Communications of the ACM*, 15(10):859–866, October 1972. Turing Award Speech.

[5] John Harrison. Formal verification at Intel. In *Proceedings of the 18th IEEE Symposium on Logic in Computer Science (LICS)*, Ottawa, Ontario, Canada, 2003. Invited talk `http://www.cl.cam.ac.uk/~jrh13/slides/lics-22jun03.pdf`.

[6] C. A. R. Hoare. The emperor's old clothes. *Communications of the ACM*, 24(2):75–83, February 1981. Acceptance speech for 1980 Turing Award.

[7] Poul-Henning Kamp. My compiler does not understand me. *Communications of the ACM*, 55(7):51–53, 2012.

[8] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, USA, October 2009.

[9] R. A. Mollin. Prime-producing quadratics. *The American Mathematical Monthly*, 104(6):529–544, 1997.

[10] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How Amazon web services uses formal methods. *Communications of the ACM*, 58(4): 66–73, 2015.

Appendix J
Jokes on Engineering Practice vs. *Theory*

This section contains a collection of stories and jokes to help you appreciate the relationship between theory and engineering practice. Generally speaking, engineers are concerned with the finite, whereas theoreticians are concerned with the infinite. Engineers build things in the material world, which is finite.

J.1 A theoreticians's salary

A theoretician is someone who does not care about their salary up to a constant factor.

A theoretician would focus on the asymptotic behaviour (*i.e.*, big-O) rather than the constant factors of a data structure or algorithm. When the inputs get large enough, the asymptotic behaviour is all that matters. But when the inputs are small, the constant factors assume greater importance.

For example, suppose the theoretician's salary grows quadratically ($O(n^2)$) in the number of years they have been working, whereas the practitioner's salary grows merely linearly ($O(n)$). The theoretician might think that they have a higher salary. The practitioner would reserve judgment until they knew the constant factors. For example, if the practitioner's salary was \$10,000 $\times n$, that would be better than a theoretician's salary of \$1 $\times n^2$ within the range of a human lifetime, because of the constant factors.

This same kind of analysis matters when selecting data structures. For example, a hash table might have $O(1)$ asymptotic lookup, but with large constant factors in both time and space. In practice, if the set is expected to contain only two or three elements, it might actually be faster and more space-efficient to use a list and do linear search. Linear search is $O(n)$, which is clearly worse than $O(1)$ in theory — and in practice for large inputs — but the constant factors matter when the inputs are small.

J.2 *British* vs. *French Engineers*

Q: What did the French engineer say to the British engineer?

A: Yes, well, it works in practice — *but does it work in theory?*

There are a few things happening in this joke. Culturally, the British approach tends to focus more on practice and less on theory, whereas the Continental European (French, German, *etc.*) approach tends to focus more on theory and less on practice.

The more typical line is *yes, it works in theory — but does it work in practice?* In normal engineering design, we first construct a theoretical model of a design, and only after analyzing that theoretical model do we build and test in the material world.

However, historians of science and engineering have demonstrated that normal engineering design does not illustrate the entire relationship between theory and practice: sometimes practice precedes, rather than follows, theory. A major case study here is thermodynamics and the steam engine. When Newcomen and Watt invented the steam engine they did not have the theory of thermodynamics. Rather, the theory of thermodynamics was invented to explain and understand their engines.

The steam engine was invented by Newcomen. Newcomen's engine had only a single chamber that was heated and cooled. Watt's engine had separate heating and cooling chambers, and so was much more efficient and powerful.

This joke reminds us that neither theory nor practice is superior to the other: we must understand both, and strive for them to work in harmony to achieve the best results.

J.3 *Engineer's Induction*

In mathematics, an inductive proof involves a base case and an inductive case. The inductive case shows that if one case is true, then the next case after it is also true. The base case is concrete (just for one particular input), whereas the inductive case is abstract (covering many inputs).

The term *engineer's induction* is a derisive term for the practice of just trying a few concrete cases in an effort to establish a general truth. For example, in 1772 Euler noticed that the following polynomial seems to always compute primes:

Actually, Euler noticed the polynomial $x^2 - x + 41$. The polynomial with the positive x term is actually due to Legendre in 1798, but nowadays everyone refers to it as Euler's.

R. A. Mollin. Prime-producing quadratics. *The American Mathematical Monthly*, 104(6):529–544, 1997

$$P(x) = x^2 + x + 41$$

Does $P(x)$ always compute a prime, for any input x? Let's use engineer's induction:

x	$P(x)$	Prime?
0	41	✓
1	43	✓
2	47	✓
3	53	✓
4	61	✓
5	71	✓

So, by engineer's induction we conclude that the polynomial $P(x)$ always produces a prime number.

But is our conclusion really sound? No. Can you guess what input does not produce a prime? Good engineering intuition suggests we try 41, since it is a named constant in the formula. And indeed, that does not produce a prime, which we can see because 41 would be a common factor:

$$P(41) = 41^2 + 41 + 41 = 41 \times (41 + 1 + 1)$$

Engineer's induction is a quick-and-dirty technique that can be useful as a first-order approximation, but it should not be considered as proof.

All of software testing is engineer's induction. In testing all we do is try some specific inputs. We hope that other inputs, that we have not tested, will also produce the desired results, but we have no reason to believe that they will other than engineer's induction. In this course you will learn some more powerful ways to think about software testing (which is very important for compilers), but they are still ultimately engineer's induction.

In the famous words of Edsgar Dijkstra, *program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence*.

Edsgar W. Dijkstra. The humble programmer. *Communications of the ACM*, 15(10):859–866, October 1972. Turing Award Speech

In your future careers you will probably use more powerful techniques for establishing program correctness called *formal methods*. Formal methods involves using mechanized logic tools to give higher confidence that software (or algorithms) are correct.

The computer hardware engineering community has been using these in practice for over twenty years, since the 1994, when Intel[1] had to recall Pentium chips with faulty floating point units. That recall cost them about $500M. It is less expensive to invest the effort to use formal methods up front to verify a hardware design than to recall a chip.

The computer software industry, in certain focused areas, has been using formal methods since around 2005 when Microsoft[2] required device drivers to pass their Static Driver Verifier before being included with Windows. More recently, Amazon Web Services[3] has been using formal methods to verify their distributed algorithms. Concurrent software is very difficult to test, because not only does one need to establish correctness in the single-threaded case, but also

[1] John Harrison. Formal verification at Intel. In *Proceedings of the 18th IEEE Symposium on Logic in Computer Science (LICS)*, Ottawa, Ontario, Canada, 2003. Invited talk `http://www.cl.cam.ac.uk/~jrh13/slides/lics-22jun03.pdf`

[2] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft. In Eerke A. Boiten, John Derrick, and Graeme Smith, editors, *Proceedings of the 4th Integrated Formal Methods (IFM) Conference*, pages 1–20, Canterbury, UK, April 2004

[3] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How Amazon web services uses formal methods. *Communications of the ACM*, 58(4):66–73, 2015

when multiple threads interleave in different ways.

J.4 Close enough for practical purposes

Zeno's Paradox