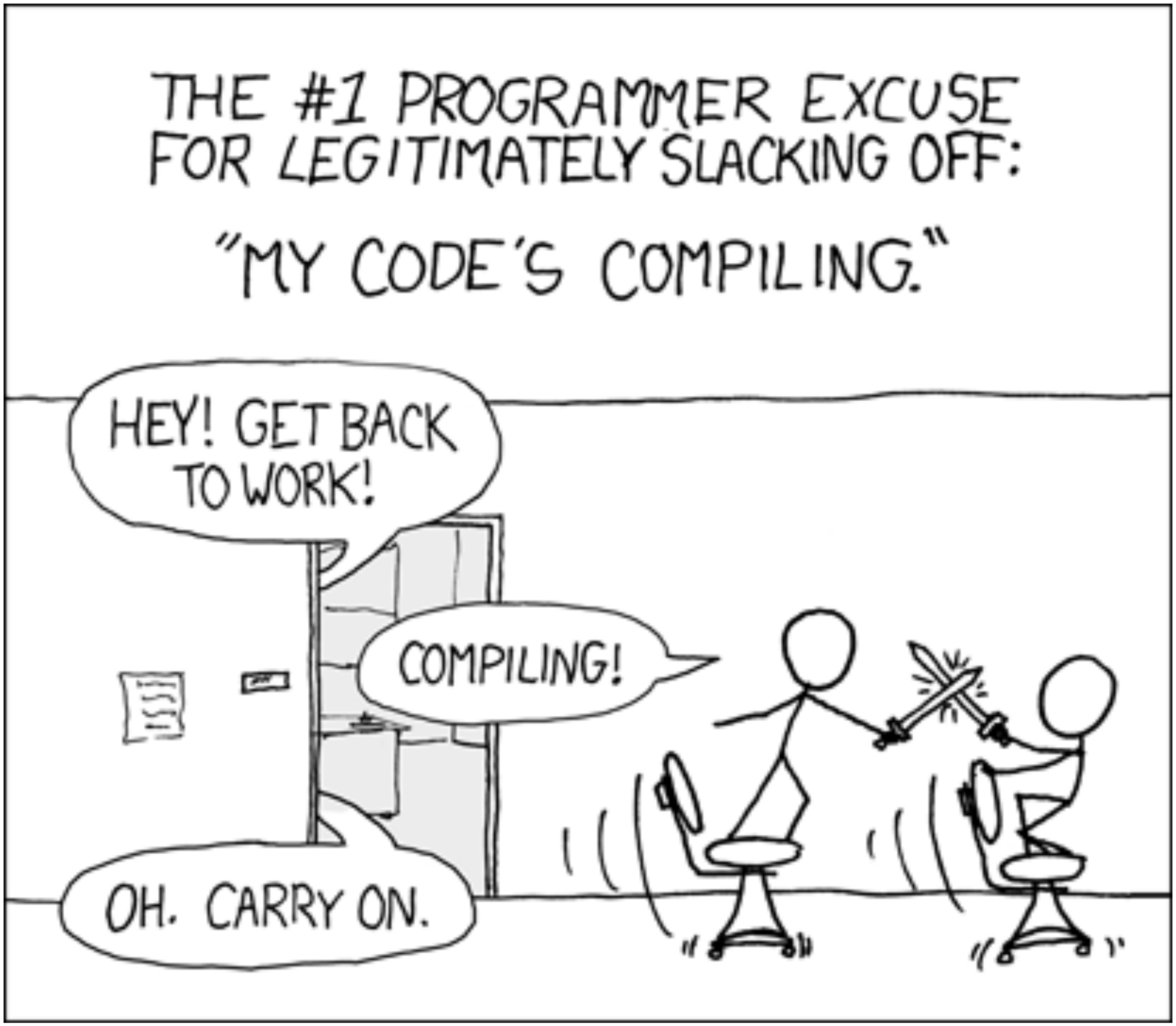
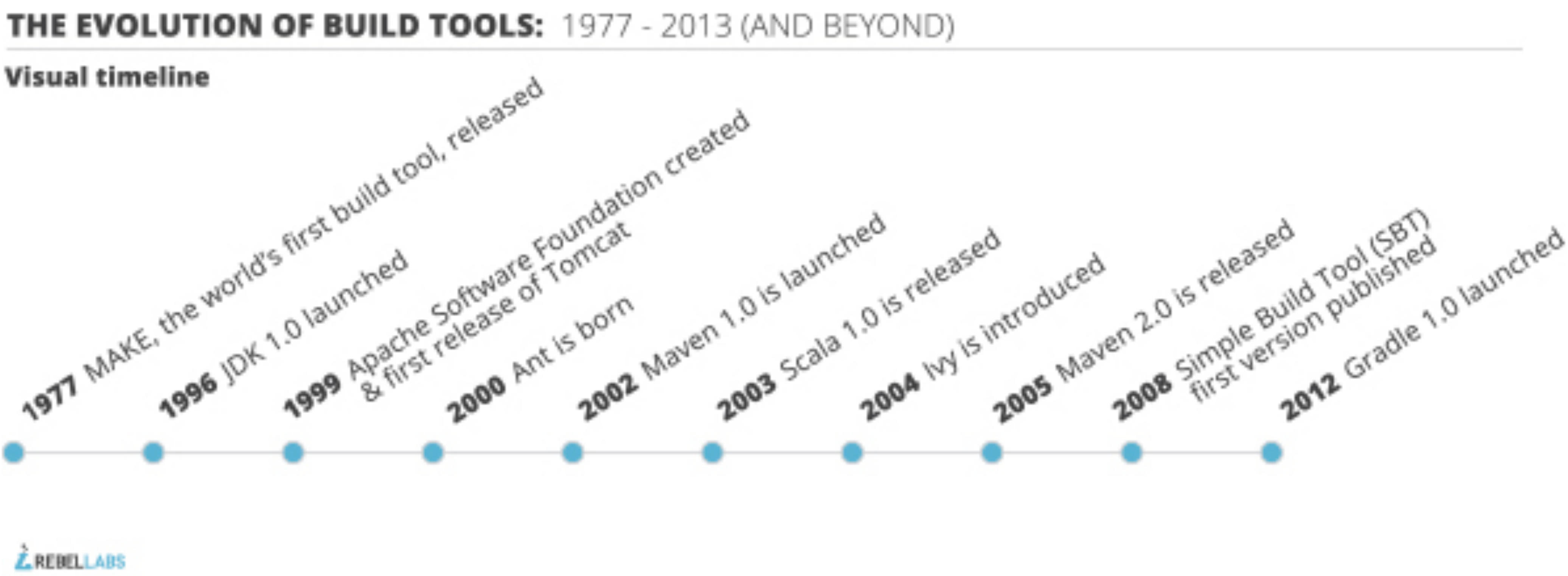
BUILD MANAGEMENT

EE 461L

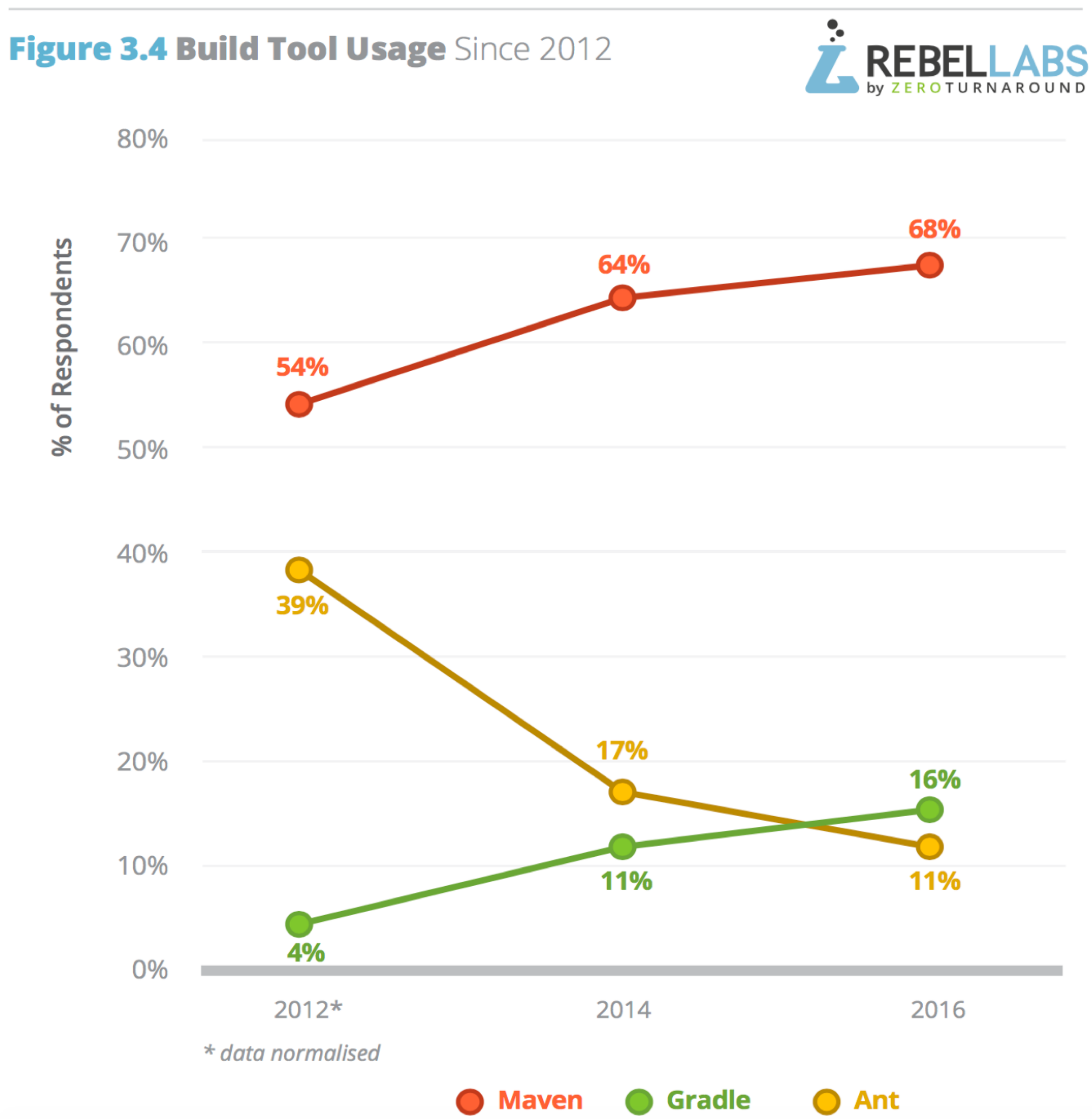
Build


THE #1 PROGRAMMER EXCUSE FOR LEGITIMATELY SLACKING OFF:
"MY CODE'S COMPILING."
HEY! GET BACK TO WORK!
COMPILING!
OH. CARRY ON.

Build Tool Evolution



Build Tool Popularity

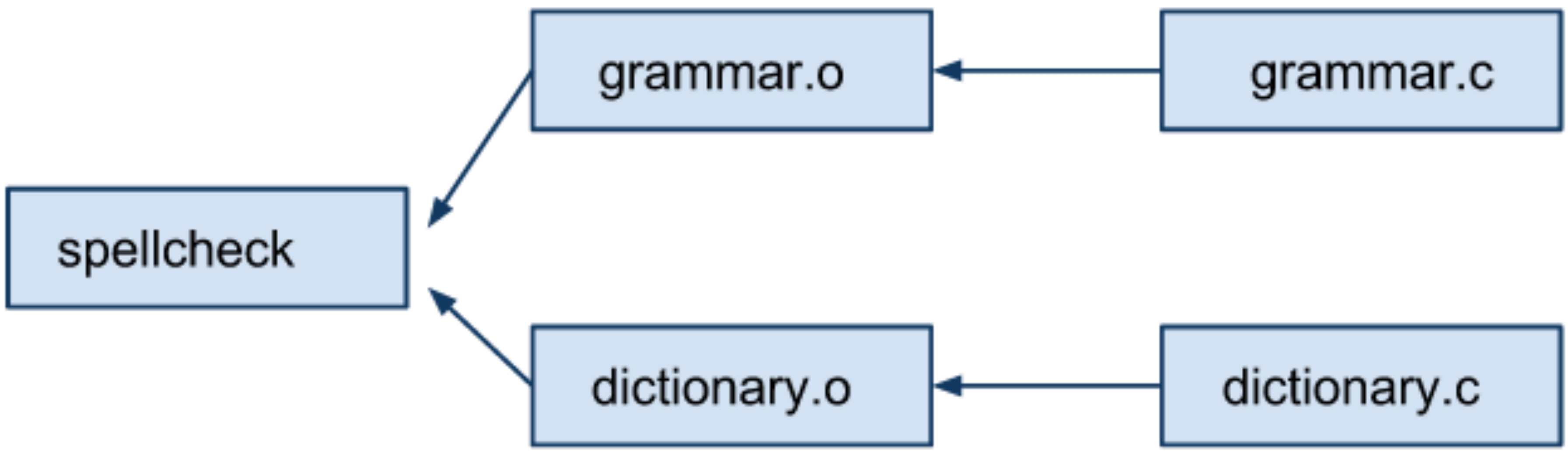


Introduction

- A program cannot just be immediately executed after a change
  - IDEs make it LOOK like this is the case, but it's not
- Intermediate steps are **required** to **build** the program
  - Convert source code to object code
  - Link objects together
- Other important aspects can also be affected by changes
  - E.g., documentation that is generated from program text
- Definitions:
  - **Source components** are those that are created manually (e.g., Java, Latex)
  - **Derived components** are those created by the computer/compiler (e.g., bytecode, PDF)

An Example

- Imagine a program `spellcheck` that consists of two components: `grammar` and `dictionary`
- Each component starts off in a C source file (`.c`)
  - Each source file is converted to an object file (`.o`) using the C compiler (e.g., `cc`)
  - Object files are linked into a final executable



Incremental Construction

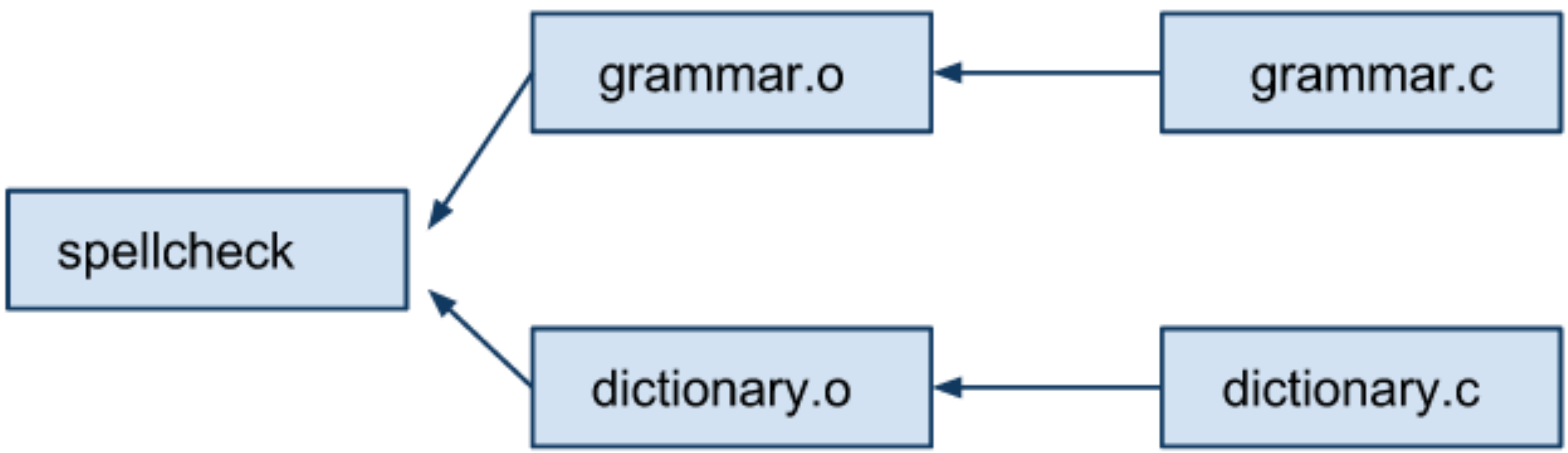
- Easiest approach: create a script
  - `cc -c -o grammar.o grammar.c`
  - `cc -c -o dictionary.o dictionary.c`
  - `cc -o spellcheck grammar.o dictionary.o`
- Problem: all steps are executed every time you run the script
  - Even if you just made changes to one of the source files
- Notice:
  - A change in component A can only impact components that are **derived** from component A

Incremental Construction (continued)

- **Rebuild Theorem**: Let A be a derived component depending on components A_1, A_2, …, A_n. The component A has to be rebuilt if:
  - A does not exist;
  - At least one A_i from A_1, A_2, …, A_n has changed; or
  - At least one A_i from A_1, A_2, …, A_n has to be rebuilt

Back to the Example…

- `grammar.o` has to be rebuilt if `grammar.c` has changed
- `spellcheck` has to be recreated because `grammar.o` has to be rebuilt

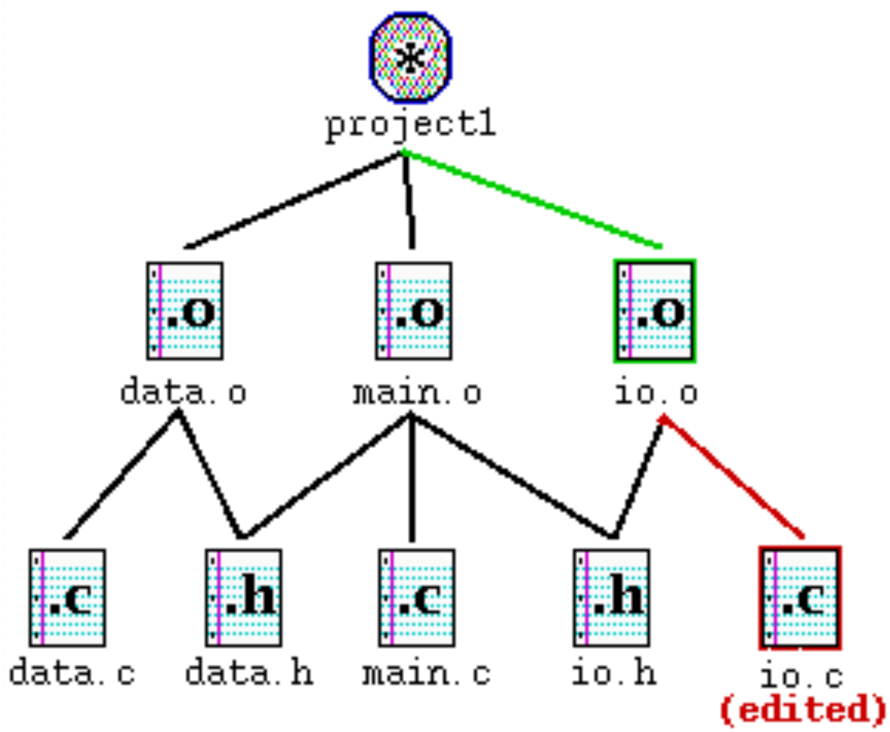


MAKE

- Created by Stuart Feldman from Bell Labs in 1975
- One of the most influential and widely used software tools
- MAKE realizes incremental program construction using a **system model**
  - A description of software product that lists individual components, their **dependencies**, and steps needed for their construction
  - MAKE's system model is specified in a file (usually called **`Makefile`**), which consists of a set of rules
  - Rules indicate component dependencies and commands needed to build components

Dependencies

- **dependency** : When a file relies on the contents of another.
  - can be displayed as a *dependency graph*
  - to build `main.o`, we need `data.h`, `main.c`, and `io.h`
  - if any of those files is updated, we must rebuild `main.o`
  - if `main.o` is updated, we must update `project1`



make demo

- **figlet** : program for displaying large ASCII text (like `banner`).
  - **http://freshmeat.sourceforge.net/projects/figlet**

- Let's download a piece of software and compile it with make:
  - download `.tar.gz` file
  - un-`tar` it
  - (optional) look at `README` file to see how to compile it
  - (sometimes) run `./configure`
    - for cross-platform programs; sets up `make` for our operating system
  - run `make` to compile the program
  - execute the program

A MAKE Rule

```
target1 target2 … targetn: source1 source2 sourcem
    command1
    command2
    …
target: source1 source2 ... sourcem
    command1
    command2
    ...
```

Example

```
spellcheck: grammar.o dictionary.o
    gcc -o spellcheck grammar.o dictionary.o
```

- The command line must be indented by a single tab
    - NOT by spaces – spaces will not work!

Running `make`

`$ make` ***`target`***

- uses the file named `Makefile` in current directory
- finds rule in Makefile for building ***target*** and follows it
  - if the ***target*** file does not exist, or if it is older than any of its *sources*, its *commands* will be executed

- variations:

`$ make`

- builds the *first* target in the Makefile

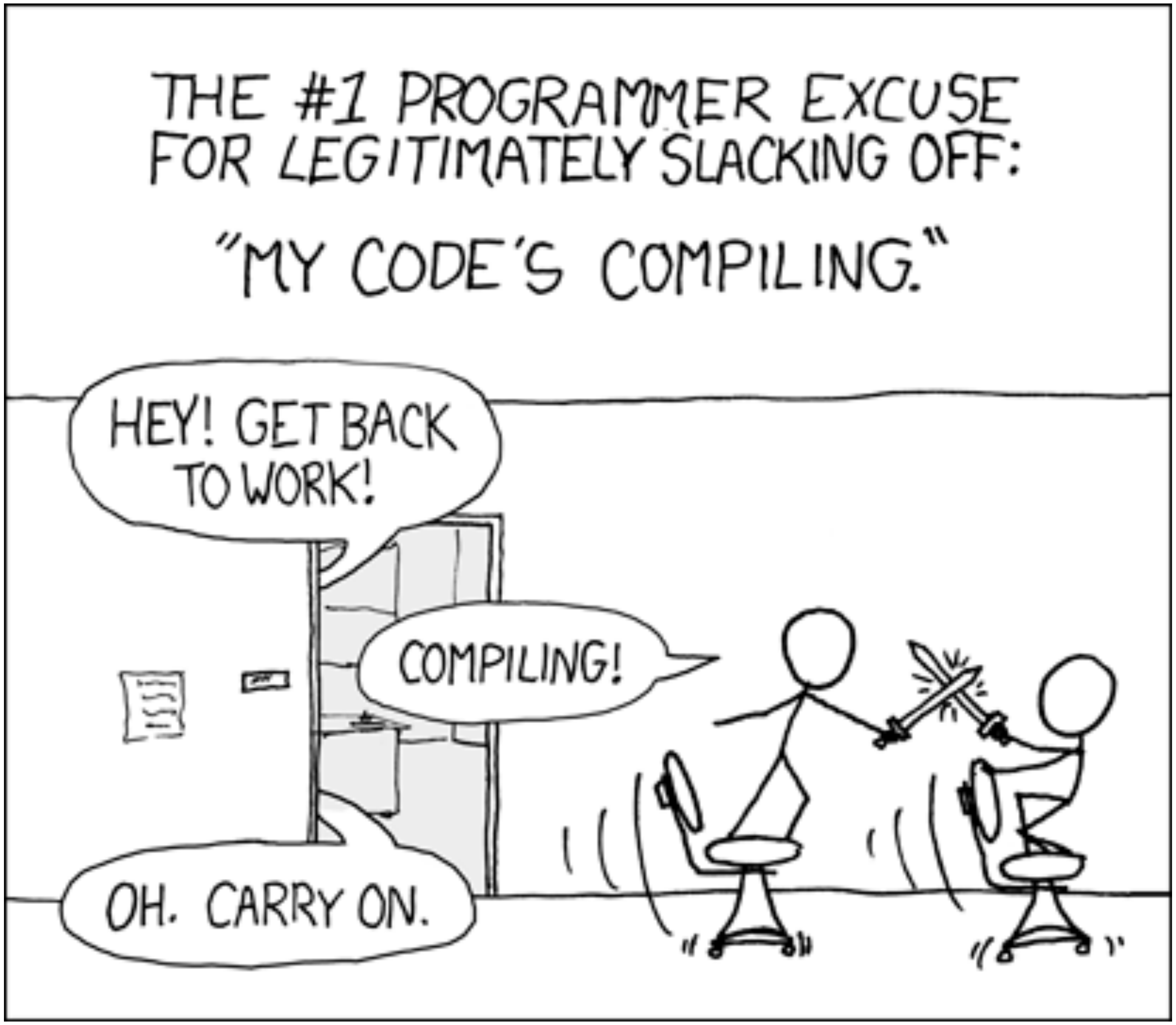
`$ make -f` ***`makefilename`***
`$ make -f` ***`makefilename target`***

- uses a makefile other than `Makefile`

Making a Makefile

- **Exercise**: Create a basic Makefile to build {hello.c, file2.c, file3.c}
  - Basic works, but is wasteful. What happens if we change file2.c?
    - everything is recompiled. On a large project, this could be a huge waste

Making a Makefile



courtesy XKCD

Making a Makefile

- Exercise: Create a basic Makefile to build {hello.c, file2.c, file3.c}
  - Basic works, but is wasteful. What happens if we change file2.c?
    - everything is recompiled. On a large project, this could be a huge waste
  - Augment the makefile to make use of precompiled object files and dependencies
    - by adding additional targets, we can avoid unnecessary re-compilation

A More Complete Example

```
spellcheck: grammar.o dictionary.o
   cc -o spellcheck grammar.o dictionary.o

grammar.o: grammar.c
   cc -c -o grammar.o grammar.c

dictionary.o: dictionary.c
   cc -c -o dictionary.o dictionary.c
```

Suppose `dictionary.c` has changed

```
$ make spellcheck
cc -c -o dictionary.o dictionary.c
cc -o spellcheck grammar.o dictionary.o
```

MAKE Algorithm

- Basic algorithm: from `Makefile` and target A_0, calculate dependency graph and run **depth first search** (DFS)
  1. Suppose A is the target component. From the graph, determine components $A_1, A_2, \ldots, A_n$ that A depends on
  2. Call algorithm on $A_1, A_2, \ldots, A_n$
  3. If one of $A_1, A_2, \ldots, A_n$ has changed or if A does not exist, rebuild A

- **Question**: how does MAKE know if a file has been changed?
  - Use time stamps from the file system

Makefile: Variables

- MAKE provides a number of properties that increase flexibility and reduce verbosity
- You can use **variables** to store values
  - VAR = value
  - Refer to variables using $(VAR) or ${VAR}

```
CC = cc
OBJECTS = grammar.o dictionary.o
spellcheck: $(OBJECTS)
	$(CC) -o spellcheck $(OBJECTS)
```

```
$ make CC=gcc spellcheck
```

More variables

```
OBJFILES = file1.o file2.o file3.o
PROGRAM = myprog
CC = gcc
CCFLAGS = -g -Wall

$(PROGRAM): $(OBJFILES)
        $(CC) $(CCFLAGS) -o $(PROGRAM) $(OBJFILES)
```

- many makefiles create variables for the compiler, flags, etc.
  - this can be overkill, but you will see it "out there"

Special variables

`$@`	the current target file
`$^`	all sources listed for the current target
`$<`	the first (left-most) source for the current target
	(there are other special variables)

```
myprog: file1.o file2.o file3.o
        gcc $(CCFLAGS) -o $@ $^

file1.o: file1.c file1.h file2.h
        gcc $(CCFLAGS) -c $<
```

- Exercise: change our hello Makefile to use variables for the object files and the name of the program

Makefile: Implicit Rules

- The construction steps for `dictionary.o` and `grammar.o` are the same; we can combine them using **implicit rules**
  - An implicit rule is applied on a number of components given via a name pattern
- A name pattern marks files through the suffix
  - `.o` contains all files that end in `.o`
  - `.c` contains all files that end in `.c`
- Implicit rule is of the form:

```
.suffix1 .suffix2:
    command
```

Makefile: Implicit Rules (continued)

```
.c.o:
    command
```

- Since implicit rules refer to many components, we need a way to refer to a component in a command
  - Implicit variable `$@` stands for the current target; `$<` stands for the left-most source

```
# conversion from .c to .o
.c.o:
    $(CC) -c $(CFLAGS) $(CPPFLAGS) -o $@ $<
```

- Rule is applied whenever MAKE searches for a target ending in `.o` and a file exists with the same basic name but ending in `.c`

Makefile: Implicit Rules (continued)

- In fact, we can avoid writing the .c.o rule altogether; it’s one of a catalog of predefined rules in MAKE
- File suffixes in a Makefile have to be defined in a special rule
- All sources of the installed target .SUFFIXES are used at the file ending for implicit rules

```
.SUFFIXES: .c .o
```

- More common to use pattern rules though

Makefile: Pattern Rules

- Pattern rules provide a more powerful alternative to file-ending (SUFFIXES) rules
- "`%`" in the target stands for arbitrary character string

```
%.o: %.c defs.h
	$(CC) -c $(CFLAGS) $(CPPFLAGS) -o $@ $<
```

- Benefits:
  - Can add additional dependencies (e.g., `defs.h` above)
  - "`%`" refers to arbitrary strings (e.g., can use to match subdirectories)

MAKE: Pseudotarget

- Does not result in a file, simply executes a number of commands

```
INSTALL_PROGRAM=install
bindir=/usr/local/bin
install: spellcheck
    $(INSTALL_PROGRAM) spellcheck $(bindir)/spellcheck
```

- Other common pseudotargets:
  - `all, uninstall, clean, distclean, dist, check, depend`

Rules with no <u>dependencies</u>

```
myprog: file1.o file2.o file3.o
	gcc -o myprog file1.o file2.o file3.o

clean:
	rm file1.o file2.o file3.o myprog
```

- make assumes that a rule's command will build/create its target
  - *but if your rule does not actually create its target, the target will still not exist the next time*, so the rule will <u>always</u> execute its commands (e.g. `clean` above)
  - `make clean` is a convention for removing all compiled files

Rules with no commands

```
all: myprog myprog2

myprog: file1.o file2.o file3.o
	gcc -o myprog file1.o file2.o file3.o

myprog2: file4.c
	gcc -o myprog2 file4.c
...
```

- `all` rule has no commands, but depends on `myprog` and `myprog2`
  - typing `make all` will ensure that `myprog`, `myprog2` are up to date
  - `all` rule often put first, so that typing `make` will build everything
- **Exercise**: add “clean” and “all” rules to our hello Makefile

MAKE: Multiple Commands

- In standard practice, MAKE uses the same command to create or update a target, regardless of which file changes
- Consider a library and replacing a portion of its code
- MAKE allows a special form of the dependency, where the action specified can differ, depending on which file has changed:

```
target :: source1
    command1
target :: source2
    command2
```

- If `source1` changes, target is created or updated using `command1`; if `source2` is modified, `command2` is used

MAKE: Recursive

- For a large project, it is common to have Makefiles for subsystems
- Example: in the main Makefile for the readline library:

```
readline:
    cd subdir && $(MAKE)
```

- This has the potential to lead to wasted calls to MAKE (as a function of dependencies)
- In practice: construct subsystems one after another according to their dependencies

Dependencies

- There are obvious dependencies (e.g., `dictionary.c` depends on `dictionary.h`)
- There are also less obvious dependencies (e.g., `dictionary.c` depends on `hash.h`)
- Other dependencies:
  - Compiler changes → all object files have to be recompiled
  - Compiler invocation changes, environment variables → recompile
- UNIX C compilers can automatically determine dependencies

```
$ cc -M grammar.c
  grammar.o: grammar.c
  grammar.o: ./hash.h
```

In Class Exercise

- For this Makefile assume the Makefile is in the same directory as a **`foo.c`**, **`bar.c`**, **`foo.h`**, **`bar.h`**, **`definitions.h`**, and **`debug.h`**

- Question 1
  - What do we see when we type "**`make CC=gcc foo.o`**" at the command line? Why?

- Question 2
  - What do we see when we type "**`make foomatic-widget`**"? Why?

What about Java?

- Create Example.java that uses a class MyValue in MyValue.java
  - Compile Example.java and run it
    - javac automatically found and compiled MyValue.java
  - Now, alter MyValue.java
    - Re-compile Example.java… does the change we made to MyValue propagate?
    - Yep! javac follows similar timestamping rules as the makefile dependencies. If it can find both a .java and a .class file, and the .java is newer than the .class, it will automatically recompile
    - But be careful about the depth of the search...
- But, this is still a simplistic feature. Ant is a commonly used build tool for Java programs giving many more build options.

ANT

ANT

- A major shortcoming of MAKE is no first class support for tasks
  - One can mimic tasks using variables and implicit rules
- Or you can use ANT, which makes tasks a first class concept
  - And comes with many predefined tasks
- ANT predefined tasks:
  - compilation: javac, jspc
  - archive: jar, zip, rpm
  - documentation: javadoc
  - file management: checksum, copy, delete, move, mkdir, tempfile
  - test: jUnit

ANT Buildfile

- A **buildfile** provides the system model, written in XML (`build.xml`)
  - One **project**: name and default **target**
  - Project: one or more targets
  - Target: name and optional dependencies
    - Coarser than MAKE targets (an ANT target refers to an activity)
    - Target can consist of a number of **tasks**

Ant

- Format of Buildfile, usually named build.xml:

```
<project>
    <target name="name">
        tasks
    </target>

    <target name="name">
        tasks
    </target>
</project>
```

- Tasks can be things like:
  - `<javac  … />`
  - `<mkdir … />`
  - `<delete … />`
  - A whole lot more… http://ant.apache.org/manual/tasksoverview.html

ANT Buildfile Example

```
<project name="SimpleProject" default="dist">
  <target name="compile">
    <mkdir dir="classes"/>
    <javac srcdir="." destdir="classes"/>
  </target>
  <target name="dist" depends="compile">
    <mkdir dir="lib"/>
    <jar jarfile="lib/simple.jar" basedir="classes"/>
  </target>
  <target name="clean">
    <delete dir="classes"/>
    <delete dir="lib"/>
  </target>
</project>
```

ANT Buildfile Example

```
$ vi HelloWorld.java
$ ant
Buildfile: build.xml
compile:
    [mkdir] Created dir: /Users/mve/temp/antexamples/classes
    [javac] Compiling 1 source file to /Users/mve/temp/antexamples/classes

dist:
    [mkdir] Created dir: /Users/mve/temp/antexamples/lib
      [jar] Building jar: /Users/mve/temp/antexamples/lib/simple.jar
BUILD SUCCESSFUL
Total time: 0 seconds

$ cd classes
$ ls
HelloWorld.class
$ java HelloWorld
Hello World
$ cd ../lib
$ ls
simple.jar
```

Underlying ANT Process

- The default target of `build.xml` is `dist`
- The target `dist` depends on `compile`, so `compile` must be realized first
- The `compile` target is realized by the two tasks `mkdir` and `javac`, which must be executed first
- Now the tasks of `dist` can follow: `mkdir` and `javac`
- All targets are realized and the build was successful

Ant Exercise

- Create an Ant file to compile our Example.java program

Ant Exercise

- Create an Ant file to compile our Example.java program

```
<project>
   <target name="clean">
      <delete dir="build"/>
   </target>

   <target name="compile">
      <mkdir dir="build/classes"/>
      <javac srcdir="src"
 destdir="build/classes"/>
   </target>
</project>
```

Ant Example

- To run ant (assuming `build.xml` is in the current directory):

  `$ ant` ***`targetname`***

- For example, if you have targets called clean and compile:

  ```
  $ ant clean
  $ ant compile
  ```

Refer to:
http://ant.apache.org/manual/tasksoverview.html
for more information on Ant tasks and their attributes.

ANT Summarized

- Like MAKE, ANT works **incrementally**
  - With every new build, only those targets are realized whose dependencies have changed
- Unlike MAKE, incrementally is not part of the tool
  - Instead it is realized by the individual **task** implementations
  - E.g., javac task determines dependencies between Java classes automatically and builds just those classes that must be reconstructed
- ANT is a framework; it can easily be extended by additional tasks
  - By subclassing the Task class
- Can configure at runtime (e.g., replace original javac)

ANT Remarks

- Huge benefit: **portability**!
  - `MAKE UNIX: rm -rf classes/`
  - `MAKE Windows: rmidir /S /Q classes`
  - `ANT: <delete dir="classes"/>`
- Use of XML, which is hierarchical, partly ordered, and pervasively cross-linked, can be a barrier to entry
- Backwards compatibility is not high
  - E.g., older tasks such as `<javac>`, `<exec>`, and `<java>` use default values for options that are not consistent with more recent versions
- Provides limited fault handling rules and no persistence of state
  - Cannot use ANT as a workflow tool for any workflow other than build and test

MAVEN

Easy to start

Gets complicated quickly

Maven Basics

- Maven is more than a build tool. It also:
  - centralizes project information
  - collects information about the software and its build
  - documents the software and the project more generally

Maven POM

- POM stands for Project Object Model
- Describes a project
  - Name and version
  - Artifact type
  - Source code locations
  - Dependencies
  - Plugins
- Uses XML by default (though not the way Ant uses XML)

Project Name (GAV)

- Maven uniquely identifies a project using:
  - **groupId**: arbitrary project grouping identifier (no spaces or colons); usually loosely based on Java package
    - Often starts with reversed domain name you control (not required)
  - **artifactId**: unique name of project (no spaces or colons)
    - name of the jar without version
    - E.g., maven, my-app
  - **version**: version of project
    - E.g., 1.0, 1.1, 1.0.1, 2.0
- The project `groupId:artifactId:version` (GAV) makes the project unique

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <version>1</version>
</project>
```

Packaging

- Build type identified using the `packaging` element
- Tells Maven how to build the project
- Example packaging types: JAR, WAR, EAR, pom
- Default is `jar`

```xml
<project>
    <modelVersion>4.0.0</modelVersion>
    <groupId>org.lds.training</groupId>
    <artifactId>maven-training</artifactId>
    <version>1.0</version>
    <packaging>jar</packaging>
</project>
```

Inheritance

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <version>1</version>
</project>
```

POM files can inherit configuration:

- groupId, version
- project configuration
- dependencies

Parent POM:

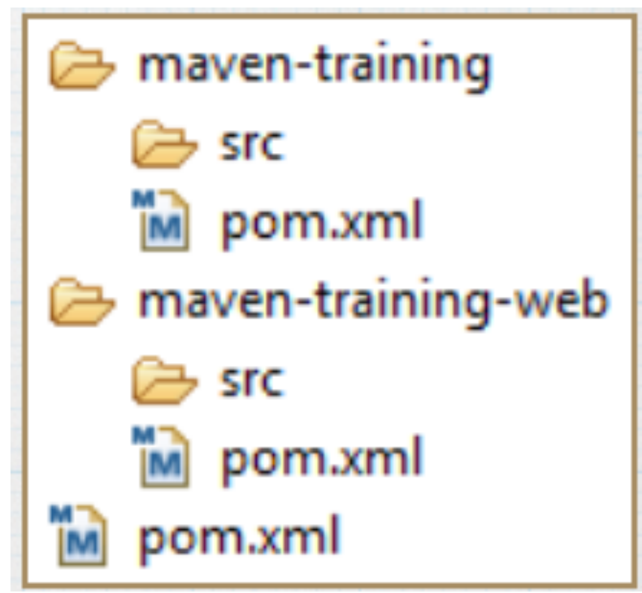
- Not distributed – only referenced from other projects
- Reference from child POM file contains GAV in parent POM file

```
<project>
  <parent>
    <groupId>com.mycompany.app</groupId>
    <artifactId>my-app</artifactId>
    <version>1</version>
  </parent>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.app</groupId>
  <artifactId>my-module</artifactId>
  <version>1</version>
</project>
```

Multi-Module Projects (Aggregation)

- Maven has 1st class multi-module support
- Each maven project creates one primary artifact
- A parent pom is used to group modules
- Modules are projects that this POM lists – are executed as a group
- `pom` packaged project: aggregates build of a set of projects by listing them as modules

```
<project>
    ...
    <packaging>pom</packaging>
    <modules>
        <module>maven-training</module>
        <module>maven-training-web</module>
    </modules>
</project>
```



Maven Conventions

- Conventions over Configurations
- Maven project structure conventions:
  - target: default working directory
  - src: all project source files
  - src/main: all sources that go in the primary artifact
  - src/test: all sources for testing the project
  - src/main/java: all java source files
  - src/main/webapp: all web source files
  - src/main/resources: all non-compiled source files
  - src/test/java: all java test source files
  - src/test/resources: all non-compiled test source files

Maven Build Lifecycle

- A Maven build follows a lifecycle
- Three built-in build lifecycles: default, clean, site
- Lifecycle consists of an ordered collection of phases

- Default lifecycle made up of these phases:
  - validate: validate project is correct and all necessary info available
  - compile: compile the source code of project
  - test: test compiled source code using unit testing framework
  - package: take compiled code and package into distributable format
  - verify: static checks, integration testing
  - install: install package into local repository (to use as dependency for other projects locally)
  - deploy:copy final package to remote repository

Example Maven Goals

To invoke a Maven build, you use lifecycle goals or phases

A phase is made up of "goals" (like tasks) - a goal is like an Ant task.

- `mvn [options] [<goal(s)>] [<phase(s)>]`

Examples:

- `mvn clean` (just invokes clean)
- `mvn clean compile` (cleans old builds, then compiles)
- `mvn clean install` (cleans, compiles, tests, packages, verifies, installs)
- `mvn test clean` (compiles, tests, then cleans)
- `mvn package` (executes all phases up through package phase)

When a phase in a lifecycle is invoked, all phases up to and including that phase are executed by Maven.

Maven and Dependencies

- Big new thing in Maven: dependency management
- Maven downloads and links the dependencies on compilation and other goals that require them
- Maven also brings in the dependencies of *those* dependencies (transitive dependencies)
- Maven repository, supported by Maven Central

Adding a Dependency

- Each dependency consists of:
  - GAV
  - Scope: compile, test, ... (default = compile)
  - Type: jar, pom, war... (default = jar)

```
<dependencies>
	<dependency>
		<groupId>junit</groupId>
		<artifactId>junit</artifactId>
		<version>4.12</version>
		<type>jar</type>
		<scope>test</scope>
	</dependency>
	...
</dependencies>
```

Maven Repositories

- Dependencies are downloaded from repositories (via http)
- Downloaded dependencies are stored in a local repository
- Repository follows a simple directory structure:
  - {groupId}/{artifactId}/{version}/{artifactId}-{version}.jar
- Maven Central is the primary community repository
  - http://repo1.maven.org/maven2

Maven Exercise

Install Maven on your computer and do the exercises on this page:

https://maven.apache.org/guides/getting-started/maven-in-five-minutes.html

Other Options

- Gradle
  - good for multi-language projects

QUESTIONS?