# Getting It Right

COMS W4115

Prof. Stephen A. Edwards
Spring 2007
Columbia University
Department of Computer Science

## Getting It Right

Your compiler is a large software system developed by four people.

How do you get it right?

## Subjects

- Team-oriented development
- Interface-oriented design
- Version control systems: CVS & Subversion
- assert()
- Regression test suites
- Writing tests
- Code coverage
- Build tools: Make & Ant
- Eclipse

## Team-oriented Development

Basic challenge: Remove as many inter-person dependencies as possible.

One group asked if the lexer/parser person should finish before the tree walker person started.

Divide and conquer: try to make it so that each person can work at his/her own rate and not depend on others.

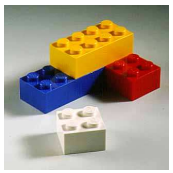
Tricky: each pass depends on the previous one.

Solution: careful design and modularity

## Interface-oriented Development

Divide your compiler into a series of modules, e.g.,

1. Lexer/Parser
2. Static semantics
3. Code generation
4. Assembler



Clearly define the interface between each module.

You'll want to write this in your project report, anyway.

Make the interfaces the "contracts" between the team members.

## Interface-oriented design

Write the interfaces first.

Document them well.

Write the public class definition, the method declarations, and the comments first.

Later, fill in code for each method, private fields, etc.

Use javadoc to extract documentation from your Java code and share with other group members

## Version Control Systems

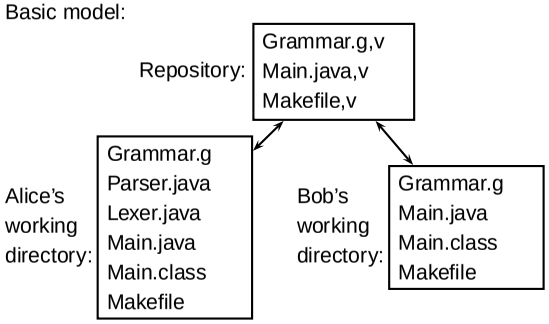
Four people working on a single program is not as easy as just one.

Need some way to make sure everybody's working on the same program.

Version control systems a good solution.

## The CVS Version Control System

Basic model:




## Using the CVS Version Control System

1. Prepare a repository
2. Add an empty sudirectory to the repository
3. Create a working directory
4. Add files, update directory, commit changes

One group member does 1,2 once.

Each group member does 3 once.

Each group member does 4 repeatedly.

## Using CVS

Creating a working directory:

```
% mkdir mydir
% cd mydir
% cvs checkout ourproj
```

Editing, adding, and updating

```
% cd ourproj
edit files, compile, etc.
% cvs add Grammar.g
% cvs commit Grammar.g
% cvs update
```

## Subversion

CVS for the '00s?

CVS plus

- Better support for renaming files and moving directories
- Can run over HTTP as well as through ssh
- Better difference computations
- Better support for binary files
- Versioning of symbolic links

## Assert

```
class Foo {
    public static void main(String[] args) {
       assert false;
    }
}
```

```
% javac -source 1.4 foo.java
% java -ea Foo
Exception in thread "main"
    java.lang.AssertionError
        at Foo.main(foo.java:3)
```

## Assert Philosophy

- Catch errors early and often
- Check function arguments are acceptable
  E.g., `assert n != null;`
- Check function return value is consistent
- Check constructor has filled in every field
- Check object state is consistent
- Check loop invariants
- For the really ambitious, write methods that check consistency of a whole data structure.

## Regression test suites

How to avoid introducing new bugs when adding features?

Partial answer: build something that tells you whether you've broken the program.

Regression suite:

- collection of tests
- exercises as much of your program as possible
- results are compared with "golden" references

## Regression tests

Easiest is when program takes a text file as input and produces text as output.

Fortunately, compilers behave like this.

Regression test inputs: short programs

Regression test golden references: assembly language

## Example tests

```
module test_emit1:
type a;
type b;
input a;
input b : integer;
output c : integer;

emit a;
emit b;
emit c

end module
```

```
module test_emit2:
output a;
output b : integer;
output c : float;

emit a;
emit b(10);
emit c(5.0f)

end module
```

## Writing Tests

Try to cover as much of your language as possible.

Try to write one test for each feature mentioned in the language reference manual.

Build sequences of tests that start with simple versions of a feature and build into the most complex.

Keep tests focused: easier to track down fault if one fails.

## Running Tests

Easiest is to use a scripting language that

- invokes the test,
- compares the outputs, and
- logs results and any errors

For CEC, I wrote a shell script to do this.

## Shell Script

Carefully runs two programs.

Compares output to reference file.

Stores results when it differs.

```
#!/bin/sh

STRLXML=./strlxml
XMLSTRL=./xmlstrl

globallog=teststrlxml.log
rm -f $globallog
error=0
```

## Shell Script

```
Check() {
  basename=`echo $1 | sed 's/.*\\///
                           s/.strl//'`
  reffile=`echo $1 | sed 's/.strl$/.out/'`
  xmlfile=${basename}.xml
  outfile=${basename}.out
  difffile=${basename}.diff
  echo -n "Parsing $basename..."
  echo "Parsing $basename" 1>&2
  $STRLXML < $1 > $xmlfile 2>&1 || {
    echo "FAILED: strlxml terminated"
    error=1 ; return 1
  }
  $XMLSTRL < $xmlfile > $outfile 2>&1 || {
    echo "FAILED: xmlstrl terminated"
    error=1 ; return 1
  }
```

## Shell Script

```
  diff -b $reffile $outfile > $difffile 2>&1 || {
    echo "FAILED: output mismatch"
    error=1 ; return 1
  }
  rm $xmlfile $outfile $difffile
  echo OK
}

for file in tests/test*.strl
do
  Check $file 2>> $globallog
done

exit $error
```

## Code coverage

Basic idea: your test suite should at least send the program counter over every part of your code.

To measure coverage, need some sort of tool that can tell when each line of code is executed.

I found a couple of them:

- gcov: works with gcc to report for C (C++?)
- clover: Commercial tool for Java, but free for students and open-source developers

There are many more.

## Example of gcov

```
$ gcc -fprofile-arcs -ftest-coverage tmp.c
$ a.out
$ gcov tmp.c
87.50% of 8 source lines executed in file tmp.c
Creating tmp.c.gcov.
```

```
        main() {
     1      int i, total;
     1      total = 0;
    11      for (i = 0; i < 10; i++)
    10        total += i;
     1      if (total != 45)
######        printf ("Failure\n");
            else
     1        printf ("Success\n");
     1    }
```

## Makefiles

How do you make it easy to compile your compiler?

Need to run ANTLR to generate files, then run javac on the results.

How do you make sure everything gets compiled when needed?

## A Basic Makefile

```
% cat Makefile
Simp.class : Simp.java
        javac Simp.java
% make Simp.class
javac Simp.java
% make Simp.class
make: `Simp.class' is up to date.
%
```

## A More Complicated Makefile

```
JAVASRC = SimpParser.java SimpWalker.java \
          SimpLexer.java SimpParserTokenTypes.java

SimpParser.class : $(JAVASRC)
        javac $(JAVASRC)

SimpParser.java SimpLexer.java : Simp.g
        java antlr.Tool Simp.g

clean :
        rm -f *.class SimpParserTokenTypes.txt \
        SimpParser.java SimpWalker.java \
        SimpLexer.java \
        SimpParserTokenTypes.java
```

## A More Complicated Makefile

```
% make
java antlr.Tool Simp.g
ANTLR Parser Generator   Version 2.7.1   1989-2000 jGuru.com
javac SimpParser.java SimpWalker.java SimpLexer.java SimpParserT
% rm SimpParser.class
% make
javac SimpParser.java SimpWalker.java
  SimpLexer.java SimpParserTokenTypes.java
% make clean
rm -f *.class SimpParserTokenTypes.txt \
SimpParser.java SimpWalker.java \
SimpLexer.java \
SimpParserTokenTypes.java
```

## Writing Makefiles

Rules take the form:

*target* : *source source ...*

*commands*

tab

Variable definition and use:

*variable* = *value*

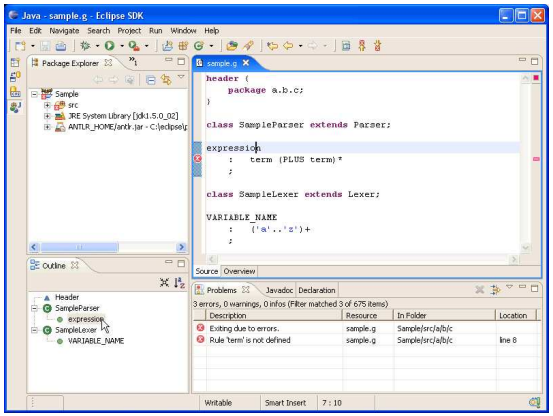
$(*variable*)

## Ant

A Java-centric build tool with instructions in “build.xml”

```
<project name="MyProject" default="dist" basedir=".">
    <description>simple example build file</description>
  <property name="src" location="src"/>
  <property name="build" location="build"/>
  <property name="dist"  location="dist"/>

  <target name="init">
    <!-- Create the time stamp -->
    <tstamp/>
    <!-- Create the build directory -->
    <mkdir dir="${build}"/>
  </target>

  <target name="compile" depends="init"
      description="compile the source " >
    <!-- Compile the java code from ${src} into ${build} -->
    <javac srcdir="${src}" destdir="${build}"/>
  </target>

  <target name="dist" depends="compile"
       description="generate the distribution" >
    <!-- Create the distribution directory -->
    <mkdir dir="${dist}/lib"/>
    <!-- Put everything in ${build} into
         MyProject-${DSTAMP}.jar -->
    <jar jarfile="${dist}/lib/MyProject-${DSTAMP}.jar"
         basedir="${build}"/>
  </target>

  <target name="clean" description="clean up" >
    <delete dir="${build}"/>
    <delete dir="${dist}"/>
  </target>
</project>
```

## Eclipse




## Eclipse

http://eclipse.org/

Java-based integrated development environment platform.

Java support strongest, other languages included.

Support for CVS & Subversion version-control systems.

Plug-in for ANTLR:

http://antlreclipse.sourceforge.net/