

Getting It Right

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

How do you get it right?

Getting It Right

COMS W4115

Prof. Stephen A. Edwards

Fall 2006

Columbia University

Department of Computer Science

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.

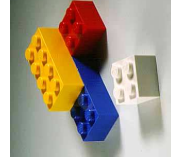
Tricky: each pass depends on the previous one.

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.

Subjects

- Team-oriented development
- Interface-oriented design
- Version control systems
- assert()
- Regression test suites
- Writing tests
- Code coverage
- Makefiles

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.

Using the CVS Version Control System

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

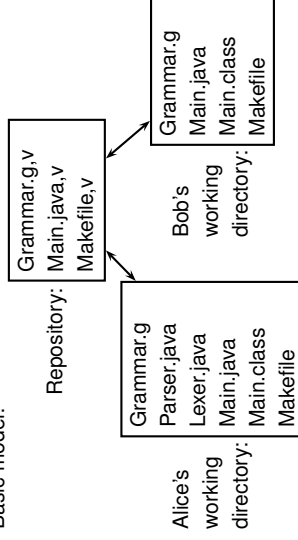
One group member does 1,2 once.

Each group member does 3 once.

Each group member does 4 repeatedly.

The CVS Version Control System

Basic model:



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
```

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:      module test_emit2:
type a;                output a;
type b;                output b : integer;
input a;               output c : float;
input b : integer;
output c : integer;    emit a;
                       emit b(10);
                       emit c(5.0f)
emit a;
emit b;
emit c                  end module
                       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=teststrl.xml.log
rm -f $globallog
error=0
```



