Getting It Right

Stephen A. Edwards

Columbia University

Fall 2010

Getting It Right

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

How do you get it right?

Subjects

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

Three-goal Structure

Think of your project as three goals:

1. Enough features so that Edwards won't flunk you
2. Enough so that you will get a reasonable grade
3. Everything you would like to do given the time

(Thanks to B. Stroustrup)

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 static semantics 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.

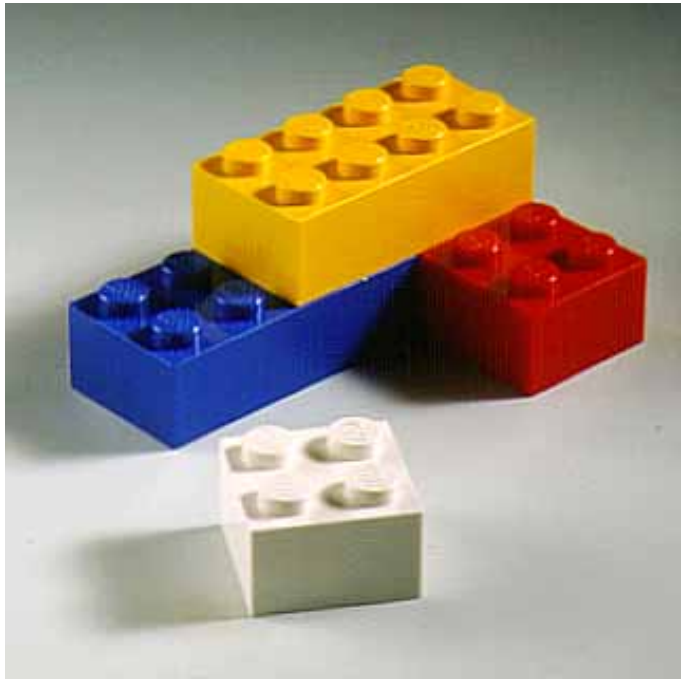
Tricky: each pass depends on the previous one.

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 (.mli files).

Write the types, the type signatures of the functions, and the comments first.

Document them well.

Later, fill in code (.ml files).

Use *ocamldoc* to extract documentation from your 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.

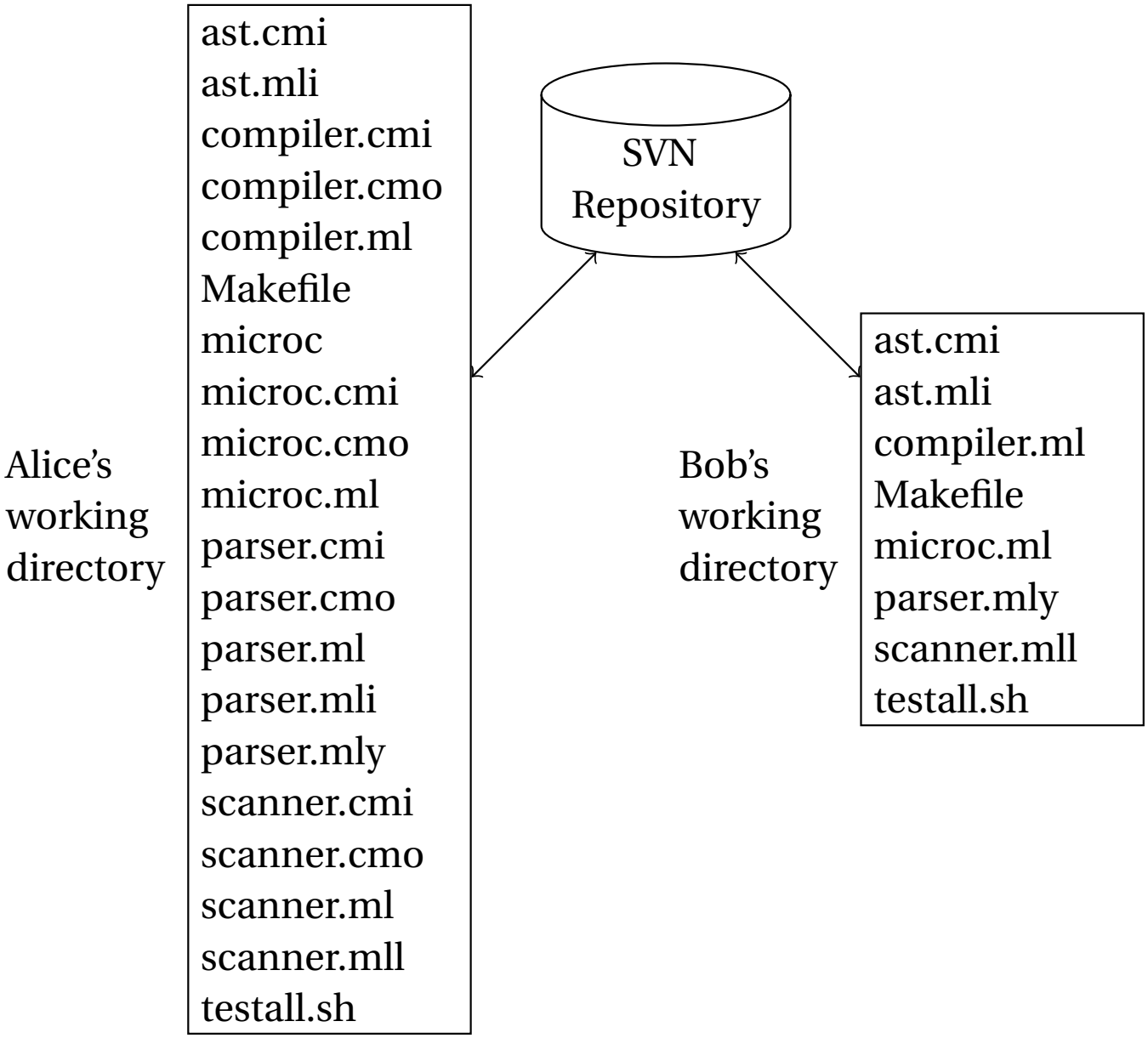
Email alone is not going to cut it.

Version control systems a good solution.

Subversion is a popular, widespread version control system.

Google code, Sourceforge, and others are free, public repositories.

The Subversion Version Control System



Using the SVN 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.

http://svnbook.red-bean.com/

Using SVN

Creating a working directory:

```
% svn checkout http://awk.googlecode.com/svn/trunk awk
```

Editing, adding, and updating

```
% cd awk
edit files, compile, etc.
% svn add parser.mly
% svn commit -m "Initial version" parser.mly
% svn update
```

Assert

assertfailure.ml:

```
assert ( 1 + 2 = 3 ) ;
assert ( 3 + 4 = 9 ) ;;
```

```
$ ocamlc -o assertfailure assertfailure.ml
$ ./assertfailure
Fatal error: exception Assert_failure("assertfailure.ml", 2, 0)
```

Checks that its argument evaluates to true; if not, throws an Assert_failure exception with the file, line, and character where the assertion failed. Otherwise, returns ().

Assert Philosophy

- Catch errors early and often
- Check function arguments are acceptable
  E.g., `assert (n > 2)`
- Check function return value is consistent
- Check object state is consistent
- Check loop invariants
- For the really ambitious, write functions 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.

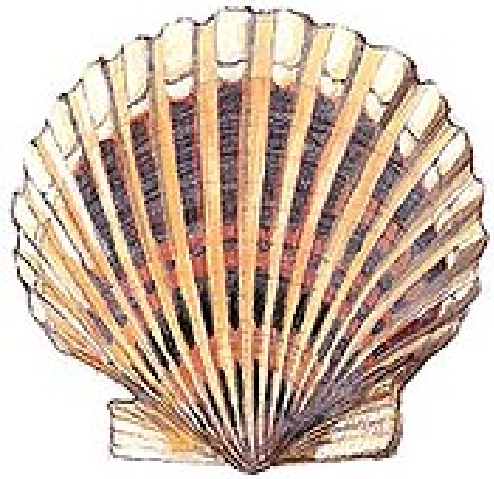
Compares output to reference file.

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 to 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 many of them:

- Bisect: works on OCaml
- 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 *ocamllex* and *ocamlyacc* to generate files, then run *ocamlc* on the results in the right order.

How do you make sure everything gets compiled when needed?

A Basic Makefile

```
% cat Makefile
simp.cmo simp.cmi : simp.ml
        ocamlc -c simp.ml
% make simp.cmo
ocamlc -c simp.ml
% make simp.cmo
make: 'simp.cmo' is up to date.
%
```

A More Complicated Makefile

```
OBJS = parser.cmo scanner.cmo printer.cmo interpret.cmo microc.cmo

microc : $(OBJS)
        ocamlc -o microc $(OBJS)
scanner.ml : scanner.mll
        ocamllex scanner.mll
parser.ml parser.mli : parser.mly
        ocamlyacc parser.mly

%.cmo : %.ml # Pattern matching
        ocamlc -c $<
%.cmi : %.mli
        ocamlc -c $<

.PHONY : clean
clean :
        rm -f microc parser.ml parser.mli scanner.ml \
        testall.log *.cmo *.cmi

# Generated by ocamldep *.ml *.mli
interpret.cmo: ast.cmi
interpret.cmx: ast.cmi
microc.cmo: scanner.cmo parser.cmi interpret.cmo
microc.cmx: scanner.cmx parser.cmx interpret.cmx
parser.cmo: ast.cmi parser.cmi
parser.cmx: ast.cmi parser.cmi
```

Running the More Complicated Makefile

```
$ make
ocamlyacc parser.mly
ocamlc -c ast.mli
ocamlc -c parser.mli
ocamlc -c parser.ml
ocamllex scanner.mll
49 states, 1995 transitions, table size 8274 bytes
ocamlc -c scanner.ml
ocamlc -c printer.ml
ocamlc -c interpret.ml
ocamlc -c microc.ml
ocamlc -o microc parser.cmo scanner.cmo printer.cmo \
  interpret.cmo microc.cmo
$ rm microc.cmo
$ make
ocamlc -c microc.ml
ocamlc -o microc parser.cmo scanner.cmo printer.cmo \
  interpret.cmo microc.cmo
$ make clean
rm -f microc parser.ml parser.mli scanner.ml \
   testall.log *.cmo *.cmi
```

Writing Makefiles

Rules take the form:

```
target : source source ...
    commands
```

tab

Variable definition and use:

```
variable = value
$(variable)
```

Eclipse

There are OCaml support packages for Eclipse; I haven't used them.

OcaIDE looks nice: