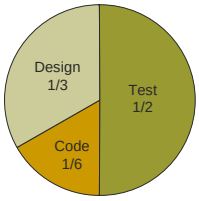
Compilers in Real Life

Dave Mandelin
2 Dec 2004

Software Development Time

From *The Mythical Man-Month* by Fred Brooks



- Can we do more error checking and less testing?
- Better yet, can we avoid writing bugs in the first place?

Software Maintenance

- Maintenance is
  - Fixing bugs
  - Enhancing functionality
  - Improving performance
  - Refactoring
- 60/60 Rule
  - 60% of project cost is maintenance
  - 60% of maintenance is enhancements
  - 30% of maintenance cost is reading existing code
  - From *Facts and Fallacies of Software Engineering* by Robert Glass

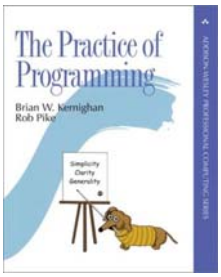
Lessons from Real Life

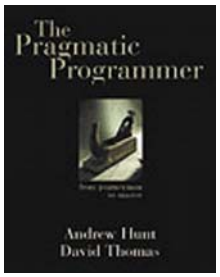
- Software needs to be
  - Reliable
  - Maintainable
  - Understandable
  - …especially if it's any good.

Solutions for Real Life

- How can we write reliable, maintainable, understandable software?
- Design a new language!
  - A language specially designed to handle ***your*** problem
  - Program is short, focused on task
  - "Junk" implementation details hidden
    - And maintainable in one place
  - Error checking
  - Error avoidance

Celebrity Endorsements



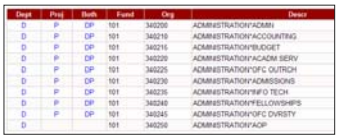


Compilers are Software

- Programming language tools need to be maintainable, understandable too
  - Compilers, code analyzers, debuggers
- We could design special languages to help implement our languages
  - Too much for most projects
    - Can be done, though (PA3, yacc)
- Focus on **simplicity** instead

Case Study 1: Search Results

Project Search



Department Search

The Problem

- Many search types
- Want same look and feel for all
  - Easy to learn, use, and understand
- Need different result format
  - Different titles, links

Solution 1: Spaghetti code

```
if (type == PROJECT) {
    link1 = “project.asp?” + name;
    link2 = “grant.asp?” + id;
} else {
    link1 = “dept.asp?” + id;
    link2 = null;
}
…
System.out.println(link1);
if (type == PROJECT) {
    System.out.println(link2);
}
```

- Maybe it works, maybe you get fired
- Unmaintainable

Solution 2: Write it over and over

- Write each search page as a separate class
  - Maybe Alice does departments, Bob does projects, …
- Hard to keep consistent look and feel

Solution 3: Recipes

- Write each search page as a separate class
  - Follow a fixed recipe each time
  - Example: recursive descent parsing
    - Follow a fixed recipe for each production
- Good strategy
  - But not the best!

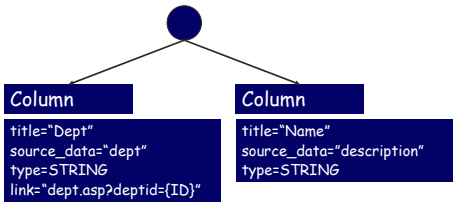
Recipes

- What's good about recipes?
  - Figure out how to do it only once
  - Avoid bugs if the recipe is correct
- What's wrong with recipes?
  - Type it in many times
  - Can type in bugs each time
  - Boring

A Better Way

- Factor out the repetition
  - Describe the differences with a *notation*
    - PA3: grammar file
    - Search: describe result format
  - Implement the repeated parts with interpreters, compilers, and libraries
    - PA3: parsing engine, table generator
    - Search: interpreter

RFL: Result Format Language

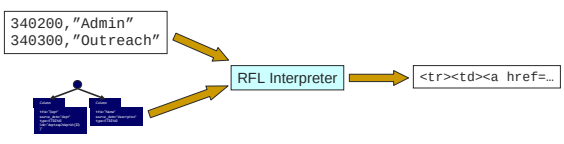


Report Format Language

- A *configuration language* for reports
- Syntactic sugar for the recipe code
- Raises level of *abstraction*
  - Java has abstraction features, too
    - methods, classes
    - Sometimes Java is not good enough
      - PA3: parsing table is unreadable
    - Need a new language

RFL Interpreter

- Search results come from database
- RFL program is an AST
  - Created programmatically – no front end
- Run RFL program on each result tuple



RFL Interpreter

- Allowed rapid development of many search pages
- One day, a user sends an email…
  - Site is slow when displaying 5000 search results
    - Don't ask
  - What can we do?

Running RFL

- Interpreter

```
for col in columns
   // Visit each column
   Object data =
        row.getData(col.name);
   String s = col.format(data);
   if (col.hasLink()) {
        col.writeLink(row);
   }
   print(s);
   if (col.hasLink()) {
        print("</a>");
   }
```

- Hand-written

```
// First column
data = row.getData("name");
s = col.format(data);
col.writeLink(row);
print(s);
print("</a>");

// Second column
data = row.getData("title");
s = col.format(data);
print(s);
```

RFL Compiler

- a.k.a. code generator
- Compile ASTs to HLL code (VBScript)
- Performs easy optimizations
  - Loop unrolling
  - Constant propagation
    - Easy because compiler knows which assignments it is generating
- 10x speedup

Expressiveness

Expressiveness, Maintenance Effort ↓

- Configuration languages
  - RFL
- Little languages
  - make, PA2 lexer spec
- Domain-specific languages (DSLs)
  - VHDL, PostScript, UnrealScript
- General-purpose languages (GPLs)
  - Java, Perl, Decaf

Implementation Performance

Execution Speed, Development Effort ↓

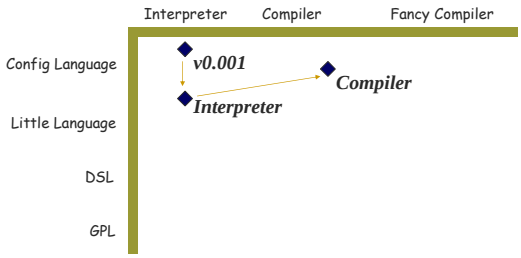
- Interpreter
  - RFL Interpreter
- Basic Compiler
  - PA4
- Optimizing Compiler
  - javac, RFL Compiler
- Fancy Optimizing Compiler
  - PA6, gcc

Usability

Usability, Language Design Effort ↓

- Author
  - one-off code generators
- Hackers
  - RFL, X configuration scripts
- Programmers
  - Java, Perl, Decaf
- Users
  - UnrealScript

Evolution of RFL



Case Study 2: Little Reports

Revenue	60,000
Deferred Revenue	14,000
Expenses	70,000
Profit	4,000

CBL: Cash Balance Language

- 'Profit'=GROUP((REV + DEF) + EXP)
  - 'title'=GROUP(…) is CBL syntax
  - REV
    - Like a primitive zero-argument function
    - Evaluated using a database query
    - What happens if we need values from a web site?
      - Need extensibility
      - CBL has an interface for implementing new primitives by writing a simple class

Error Checking in CBL

- Debits and credits are confusing
  - Which is right, REV – DEF or REV + DEF?
  - "That's like asking the square root of million. No one will ever know." – Nelson Muntz
- A type system
  - Two types: UP and DOWN
  - Same types must add, different types must subtract
  - Can check this statically
  - Is there a better way?

Error Avoidance in CBL

- Just type REV ± DEF
- CBL figures out the right operation
- Program is *underconstrained*
- Language implementation uses *inference* to select operations

CBL Implementation

- Like PA1-PA3, but simpler
  - Hand-written DFA lexer
    - I didn't have a lexer generator
  - Hand-written recursive descent parser
    - Works well for little languages
  - Interpreter
    - Operation inference
    - Expression evaluator
    - Extension interface

CBL In Practice

- I developed it in a few days
  - It was easy after PA1-PA3
- Gave the code to another CS 164 graduate, who
  - Added some new features
  - Started writing programs
- Users ask for a new report
  - It's done in 60 seconds

CBL Evaluation

- Much better than RFL
- Text-based language
- Error avoidance
- Maintainable implementation

Break

- After the break…
  - DSLs for game programming

Case Study 3: UnrealScript

The Unreal Engine

- The **Unreal engine** is the game engine which powered *Unreal*, and many more since.
  - Unreal, Unreal 2, UT, UT 2003, UT 2004, Deep Space 9: The Fallen, Deus Ex, Deus Ex: Invisible War, Postal 2, Duke Nukem Forever, …
- Since it was as customizable as Quake and featured its own scripting language **UnrealScript**, it soon had a large community on the internet which added new modifications to change or enhance game play.

From http://en2.wikipedia.org/wiki/Unreal

Customizing Games

- Unreal and similar games
  - Multiplayer simulations on the Internet
    - Unreal Tournament 2004, EverQuest 2, The Sims Online
  - Customers expect to be able to download new characters, levels, game types, and to make their own
  - Are customers going to write 10k lines of C to add a surprise birthday party to The Sims?
  - In-house game designers don't necessarily want to use C either

Customizing Games

- Game-specific programming concepts
  - Independent actors
    - E.g., person, car, elevator
      - Sounds like a Java class
      - Or it is a thread? And can we have 10k threads?
    - Have behavior
      - Java methods, sounds OK
    - Behavior depends on current state
      - Class or methods change over time? Can't do that!
  - Events, duration, networking

UnrealScript

- Design Goals
  - From http://unreal.epicgames.com/UnrealScript.htm
  - Directly support game concepts
    - Actors, events, duration, networking
  - High level of abstraction
    - Objects and interactions, not bits and pixels
  - Programming simplicity
    - OO, error checking, GC, sandboxing
- Several architectures were explored and discarded
  - Java: too slow (Java 1.1)
  - VB-based language: C programmers didn't like it

UnrealScript

- Looks like Java
  - Java-like syntax
  - Classes, methods, inheritance
- Game-specific features
  - States, networking
- Runs in a framework
  - Game engine sends events to objects
  - Objects call game engine for services

Actor States

```
void spokenTo(Speaker s) {
    if (state == ANGRY) {
        shootAt(s);
    } else {
        sayHi(s);
    }
}

void bumpsInto(Object obj) {
    backUp();
    say("Raaaaaaargh!!!");
    state = ANGRY;
}

// And what about inheritance?
```

```
state angry {
begin:
    say("Raaaaaaargh!!!");

    void spokenTo(Speaker s) {
        shootAt(s);
    }
}

void bumpsInto(Object obj) {
    backUp();
    GotoState('angry');
}

void spokenTo(Speaker s) {
    sayHi(s);
}
```

Networking

- Unreal network architecture
  - Server maintains simulation objects
  - Client also maintains simulation objects
  - Server replicates simulation objects to client
    - Sends copies of as many objects as bandwidth allows
  - Client also predicts object changes
    - Hides latency
- Language Support
  - `simulated` keyword
  - Indicates a function that can run in client prediction

Errors in UnrealScript

- Static checking
  - UnrealScript supports traditional static checking
    - Just like PA4, PA5, Java
  - Name checking
  - Type checking
- Dynamic techniques

Dynamic Error Handling

- Null pointer dereference
  - Not a problem!
    - Or not a bad problem, anyway
  - Raise an exception, return to framework
  - One event fails, the system survives
- Infinite loops and infinite recursion
  - Hard for game engine to recover from
  - `singular` function declaration
    - Means "don't recur into me"
    - Declare bugs out of existence

Language Flexibility

Configuration languages
RFL, X config scripts
Little languages
CBL, make
Domain-specific languages (DSLs)
UnrealScript, VHDL
General-purpose languages
Perl

Performance

- Implementation
  - Compiles to VM bytecode (like Java)
- Performance
  - 20x slower than C
    - Ugh! Even Java is only 2-4x slower.
    - But wait…
  - Even with 100s of objects CPU spends only 5% time running UnrealScript
  - Engine does most of the work
  - Doesn't need to be fast

Implementation Quality

Interpreter
PA1, CBL, RFL Interpreter
Bytecode Interpreter
UnrealScript, Java 1.0
Basic Compiler
PA5
Optimizing Compiler
RFL Compiler
Fancy Optimizing Compiler
Java 1.5 HotSpot VM, gcc, PA6

The Unreal Engine

- Why was it so successful?
  - Many reasons
- From a language point of view
  - Domain-specific concepts
    - Easy to use
  - Based on existing languages
    - Easy to learn
  - Runs slow
    - Easy to implement

Language Flexibility

Configuration languages
Report Format Language
Little languages
CBL: easy development and maintenance
DSLs
UnrealScript: high abstraction, easy to use
GPLs
Perl: high abstraction, years of development

Implementation Quality

Interpreter
CBL: quick development
Bytecode Interpreter
UnrealScript: slow language, fast library
Basic Compiler
PA5
Optimizing Compiler
Report Format Lang: a few simple optimizations
Fancy Optimizing Compiler
Java 1.4 HotSpot VM, gcc

Usability

Author

Sometimes appropriate: PA5 tools

Hackers

X configuration scripts

Programmers

CBL

Users

UnrealScript: high level, based on popular languages

Creating Your Own Language

- CS 164
  - Report Format Language == PA1
  - CBL == PA1-PA3
  - UnrealScript == PA1-PA5
  - You have more than enough skills!
- Hard part is language design
  - Requires experience
  - So create some languages!

Getting Started

- Language Design
  - Factor out differences from stereotypical code
  - Base on existing languages
  - Extensibility is good
- Implementation
  - Interpreter
  - Compiler
    - Compile to HLL: C, Java bytecodes, CLI
- Libraries
  - Easy to make fast
  - Good libraries make a language popular
    - Java, Perl, Python