PROGRAMMING LANGUAGES LABORATORY

Universidade Federal de Minas Gerais - Department of Computer Science

INTRODUCTION TO CODE ANALYSIS AND OPTIMIZATION

PROGRAM ANALYSIS AND OPTIMIZATION – DCC888

Fernando Magno Quintão Pereira

fernando@dcc.ufmg.br

Goals of this Course

THE PRIMARY GOAL OF THIS COURSE IS TO EXPLAIN THE STUDENT HOW TO TRANSFORM A PROGRAM AUTOMATICALLY, WHILE PRESERVING ITS SEMANTICS, IN SUCH A WAY THAT THE NEW PROGRAM IS MORE EFFICIENT ACCORDING TO A WELL-DEFINED METRIC.

- There are many ways to compare the performance of programs:
  - Time
  - Space
  - Energy consumption

We will be focusing mostly in runtime

Goals of this Course

THE SECOND GOAL OF THIS COURSE IS TO INTRODUCE STUDENTS TO TECHNIQUES THAT LET THEM UNDERSTAND A PROGRAM TO A LEVEL THAT WOULD BE HARDLY POSSIBLE WITHOUT THE HELP OF A MACHINE.

- We understand programs via static analysis techniques.
- These analyses are key to enable code optimizations.
- But they also have other uses:
  - They help us to prove correctness properties.
  - They help us to find bugs in programs.

Goal 1: Program Optimization

- The main goal of the techniques that we will see in this course it to optimize programs.
- There are many, really many, different ways to optimize programs. We will see some of these techniques:
  - Copy elimination
  - Constant propagation
  - Lazy Code Motion
  - Register Allocation
  - Loop Unrolling
  - Value Numbering
  - Strength Reduction
  - Etc, etc, etc.

```
#include <stdio.h>
#define CUBE(x) (x)*(x)*(x)
int main() {
  int i = 0;
  int x = 2;
  int sum = 0;
  while (i++ < 100) {
    sum += CUBE(x);
  }
  printf("%d\n", sum);
}
```

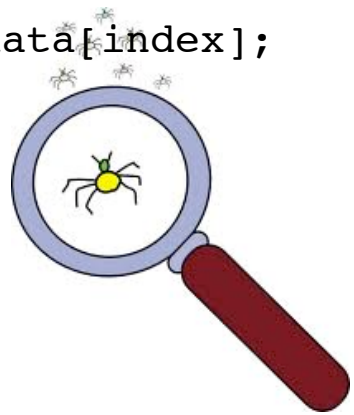
How can you optimize this program?

```
_main:
  pushl %ebp
  movl  %esp, %ebp
  pushl %ebx
  subl  $20, %esp
  call  L3
L3:
  popl  %ebx
  movl  $800, 4(%esp)
  movl  %eax, (%esp)
  call  _printf
  addl  $20, %esp
  popl  %ebx
  leave
  ret
```

Goal 2: Bug Finding

- Compiler analyses are very useful to find (and sometimes fix) bugs in programs.

```
1 void read_matrix(int* data,
       char w, char h) {
2   char buf_size = w * h;
3   if (buf_size < BUF_SIZE) {
4     int c0, c1;
5     int buf[BUF_SIZE];
6     for (c0 = 0; c0 < h; c0++) {
7       for (c1 = 0; c1 < w; c1++) {
8         int index = c0 * w + c1;
9         buf[index] = data[index];
10      }
11    }
12    process(buf);
13  }
14 }
```

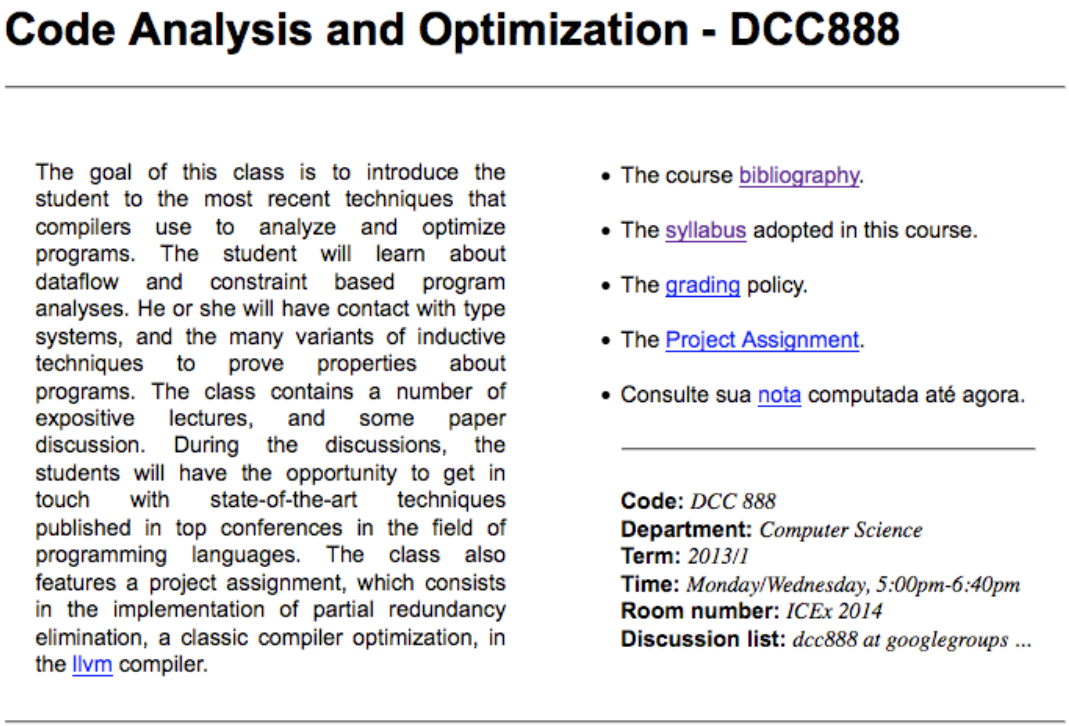


- Null pointer dereference
- Array out-of-bounds access
- Invalid Class Cast
- Tainted Flow Vulnerabilities
- Integer Overflows
- Information leaks

Can you spot a security bug in this program. **Be aware**: the bug is tricky.

The Contents of the Course

- All the material related to this course is available on-line, at http://www.dcc.ufmg.br/~fernando/classes/dcc888.
- This material includes:
  - Slides
  - Project assignment
  - Class Exercises
  - Useful links
- The page also contains the course bibliography, and a brief discussion about the grading policy



Code Analysis and Optimization - DCC888

The goal of this class is to introduce the student to the most recent techniques that compilers use to analyze and optimize programs. The student will learn about dataflow and constraint based program analyses. He or she will have contact with type systems, and the many variants of inductive techniques to prove properties about programs. The class contains a number of expositive lectures, and some paper discussion. During the discussions, the students will have the opportunity to get in touch with state-of-the-art techniques published in top conferences in the field of programming languages. The class also features a project assignment, which consists in the implementation of partial redundancy elimination, a classic compiler optimization, in the llvm compiler.

- The course bibliography.
- The syllabus adopted in this course.
- The grading policy.
- The Project Assignment.
- Consulte sua nota computada até agora.

Code: *DCC 888*
Department: *Computer Science*
Term: *2013/1*
Time: *Monday/Wednesday, 5:00pm-6:40pm*
Room number: *ICEx 2014*
Discussion list: *dcc888 at googlegroups ...*

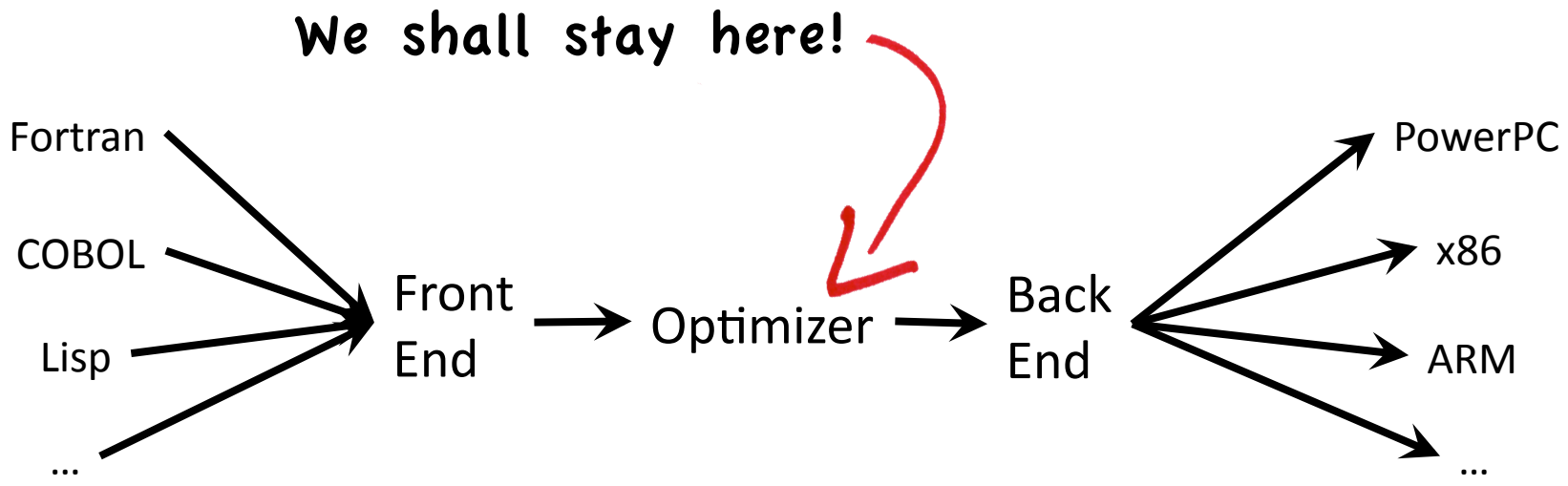
The Contents of the Course

- The course has 20 lectures. Slides for each lecture are available on-line:
  1. Introduction
  2. Control Flow Graphs
  3. Dataflow Analyses
  4. lazy Code Motion
  5. Worklist algorithms
  6. Lattices
  7. Loop Optimizations
  8. Static Single Assignment
  9. Sparse Analyses
  10. Widening and Narrowing
  11. Register Allocation
  12 . SSA-Based Register Allocation
  13. Constraint-Based Analysis
  14. Pointer Analysis
  15. Operational Semantics
  16. Type Systems
  17. Mechanical Proofs
  18. Taint Analysis
  19. Information Flow
  20. Divergence Analysis

Where we will be

- A compiler has three main parts
  - The front-end is where parsing takes place.
  - The middle-end is where optimizations and analyses take place.
  - The back-end is where actual machine code is generated.

In which CS courses can we learn about the other phases?



There is no Silver Bullet

1. It is impossible to build the perfect optimizing compiler.

- The perfect optimizing compiler transforms each program P in a program P_{opt} that is the smallest program with the same input/output behavior as P.

Can you prove the first statement?

There is no Silver Bullet

1. It is impossible to build the perfect optimizing compiler.

Let's assume that we are optimizing for size. We can reduce the problem of building the perfect compiler to an undecidable problem, such as the OUTPUT PROBLEM, e.g., "Does the program P output anything?"

The smallest program that does nothing and does not terminate is P_{least} = `L: goto L;` By definition, the perfect compiler, when fed with a program that does not generate output, and does not terminate, must produce P_{least}.

Thus, we have a decision procedure to solve the OUTPUT PROBLEM: given a program P, if the perfect compiler transforms it into P_{least}, then the answer to the problem is **No**, otherwise it is **Yes**.

The Full-Employment Theorem

1. If C is an optimizing compiler for a *Turing Complete Language*, then it is possible to build a better optimizing compiler than C.

- In other words, compiler writers will always have a job to do, as it is always possible to improve any existing compiler that compiles any meaningful programming language.

How could we prove this new theorem?

The Full-Employment Theorem

1. If C is an optimizing compiler for a *Turing Complete Language*, then it is possible to build a better optimizing compiler than C.

Lets assume that there exists the best compiler B. If P is a program, then let B(P) be its optimized version. There must be a program P_x that does not terminate, such that $B(P_x) \neq$ [L:goto L], otherwise B would be the perfect compiler. As we have seen, this is impossible.

Therefore, there exists a compiler B' that is better than B, as we can define it in the following way:

B'(P) = if P = P_x then [L:goto L] else B(P)

WHY TO LEARN COMPILERS?

DCC 888

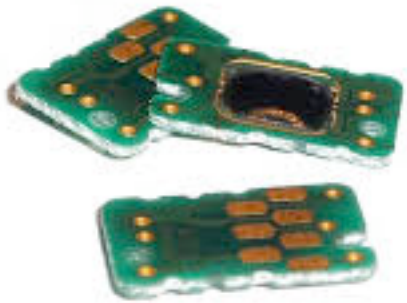
The Importance of Compilers

Robert Hundt is a leading compiler engineer working at Google. Read below what he says about the importance of compilers in general, and in that company in particular

- "At the scale of datacenters, every single performance percent matters! Just take a look at Google's (and other's) publicly available numbers on expenditures on datacenters. We are talking about billions of dollars. A single percent improvement can mean millions of dollars from more program features or improved utilization."
- "In order to deploy software at Google scale, engineers will touch a good dozen of programming and configuration languages, all with their own interesting optimization problems (from a compiler writer's point of view). Fundamental knowledge in language, compiler, and runtime implementation will help make better engineering decisions, everywhere."
- "Did you know that many of our first today's most celebrated engineers have compiler backgrounds? Jeff Dean, Sanjay Ghemawat, Urs Hoelzle, and many others. It's not a coincidence. Compiler optimization trains in big-fiddling as well as algorithmic thinking, which are essential to success in today's world."

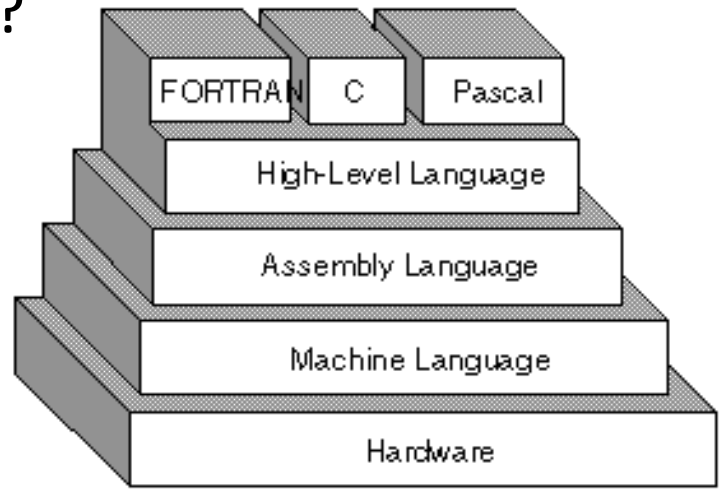
The Importance of Compilers

We have hundreds of different programming languages. How do you think they are executed?



But… can you think on why it would do good to you, from a personal point of view, to learn compilers?

And we have hundred of different hardware. How can we talk to all these different worlds?



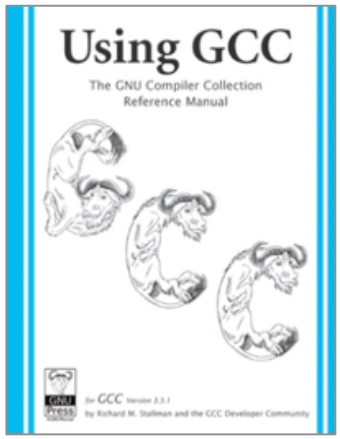
Why to Learn Compilers

- We can become better programmers once we know more about compilation technology.
- There are plenty of very good job opportunities in the field of compilers.
  - "*We do not need that many compiler guys, but those that we need, we need them badly.*"♤
- We can become better computer scientists if we have the opportunity to learn compilation technology.
  - A microcosm crowed with different computer science subjects.
  - Lots of new things happening all the time.

Is there a difference between programmers and computer scientists?

♤: quote attributed to Jean-François Bodin, professor at the University of Rennes 1. Previously senior engineer at CAPs

Compilers Help us to be Better Programmers



Compilers usually have different optimization options. The iconic gcc –O1, for instance, runs these optimizations.

What do each of these things do?

```
fauto-inc-dec
fcompare-elim
fcprop-registers
fdce
fdefer-pop
fdelayed-branch
fdse
fguess-branch-probability
fif-conversion2
fif-conversion
fipa-pure-const
fipa-profile
fipa-reference
fmerge-constants
fsplit-wide-types
ftree-bit-ccp
ftree-builtin-call-dce
ftree-ccp
ftree-ch
ftree-copyrename
ftree-dce
ftree-dominator-opts
ftree-dse
ftree-forwprop
ftree-fre
ftree-phiprop
ftree-slsr
ftree-sra
ftree-pta
funit-at-a-time
```

We can even run these optimizations individually:

```
$> gcc -fdefer-pop -o test test.c
```

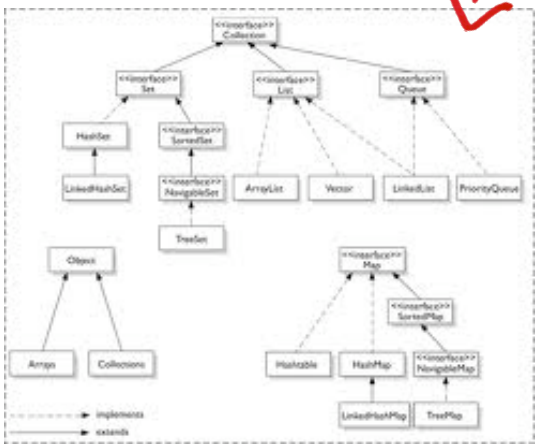
We can enable a few of them, and disable others, e.g.:

```
$> gcc -O1 -fno-defer-pop -o test test.c
```

Knowing Compilers will Fix some Misconseptions

I am a conscious programmer, because I am reusing the name 'i'. In this way my program will consume less memory.

```
int i = read();
if (i != EOF)
    i = read();
    printf("%d", i);
```



I will not use inheritance in Java, because it can make my method calls more expensive, as I have to find them in the class hierarchy.

I will use macros, instead of functions, to have a more efficient code. In this way, there will be no time lost in function calls.

What are the fallacies in these reasonings?

```
#define MAX(X, Y) (X) > (Y) ? (X) : (Y)

int max(int i, int j) { return i > j ? i : j; }
```

Lots of Job Opportunities

- Expert compiler writers find jobs in many large companies, mostly in the US, Canada, Europe and Japan: Oracle, NVIDIA, Microsoft, Sony, Apple, Cray, Intel, Google, Coverity, MathWorks, IBM, AMD, Mozilla, etc.
- These jobs usually ask for C/C++ expertise.
- Good knowledge of basic computer science.
- Advanced programming skills.
- Knowledge of compiler theory is a big plus!
- Papers published in the field will help a lot too!

compilerjobs.com

Sponsored by Nullstone Corporation - Developers of the NULLSTONE Automated Compiler Performance Analysis Suite

Compiler jobs for compiler developers who design and develop parsers, optimizers, codegenerators, assemblers, linkers, debuggers, interpreters, IDE's, and related technologies.

Compiler Writers in the Big Companies

Hardware companies need compilation technology. Either they will develop it internally, or they will buy it from third parties. This technology is also in high demand by the big internet players, which invest heavily in virtual machines.

Can you think about other examples?

Intel maintain one of the fastest C compilers, the icc, which employs state-of-the-art technology to do code optimization.

Mozilla puts a lot of effort in its JavaScript runtime environment, formed by the "Monkey" family of compilers/interpreters

LLVM, the Low-Level Virtual Machine is maintained mostly by **Apple** engineers, who keep improving it day by day.

NVIDIA has developed a entire new world of compilation technology to generate code to the highly parallel graphics processing units.

Google has an entire team of engineers working on gcc, plus some very smart people developing the V8 JavaScript engine.

Microsoft invests heavily in compilers and programming environments such as Visual Studio, and the .NET runtime environment.

Engineers from **STMicroelectronics** maintain the Open64 C compiler, which is very good at loop transformations, for instance.

Companies that Sell Compilation Technology

There are companies that sell mostly compilation technology, which can be used in several ways, e.g., to create a new back-end, to parse big data, to analyze programs for security vulnerabilities, etc.

Can you think about other examples?

Coverity is a software vendor which develops testing solutions, including static code analysis tools, for C, C++, Java and C#, used to find defects and security vulnerabilities in source code.

The Associated Compiler Experts (**ACE**) have developed compilers for over 100 industrial systems, ranging from 8-bit microcontrollers to CISC, RISC, DSP and 256-bit VLIW processor architectures.

PathScale Inc. is a company that develops a highly optimizing compiler for the x86-64 microprocessor architectures.

The Portland Group, Inc. (**PGI**) is a company that produces a set of commercially available Fortran, C and C++ compilers for high-performance computing systems.

Green Hills produces compilers for C, C++, Fortran, and Ada. The compilers target 32- and 64-bit platforms, including ARC, ARM, Blackfin, ColdFire

Compiler Jobs

LLVM backend team at Apple is looking for exceptional compiler engineers. This is a great opportunity to work with many of the leaders in the LLVM community. If you are interested in this position, please send your resume / CV and relevant information to [redacted] We are focused on improving the user experience by reducing compile time as well as maximizing the execution speed of the code generated for the Apple systems. As a key member of the Apple Compiler Team, you will apply your strong state-of-the-art background and experience toward the development of fast highly optimized compiler products that extract top performance from the Apple systems. You will join a small team of highly motivated engineers who build first-class open-source compiler tools and apply them in innovative ways.

Required Experience:

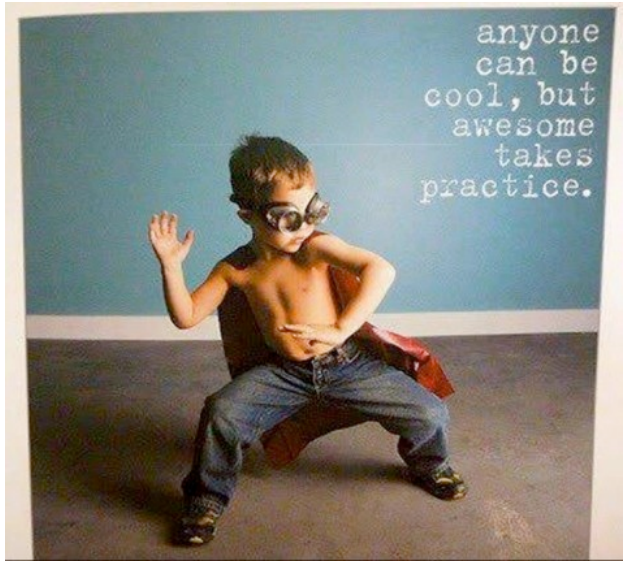
- Ideal candidate will have experience with the LLVM, GCC, or other open source / commercial compilers.
- Strong background in compiler architecture, optimization, code generation and overall design of compilers.
- Experience with developing optimizing compilers for modern architectures.
- Familiarity with analyzing generated code for optimization/code generation opportunities.
- Strong track record of building high performance, production quality software on schedule.
- Strong communication and teamwork skills.

Additional Requirements:

- Experience with developing compilers and tools for embedded devices
- Experience with developing compilers for novel micro-architectures and instruction sets
- Background in runtime compilation technologies for graphics such as OpenGL and Direct3D
- Knowledge and experience with compiler vectorization technologies

Compiler Knowledge is Fun (and Awesome)

- Compilers are very complex systems. It really takes a lot of programming skills to write them.
  - Large, robust system, usually coded in a high-performance language, such as C or C++, with lots of interactions with the operating system and the hardware.
- Compilers are backed up by a lot of computer science theory. It is not only programming.
  - Type systems, parsing theory, graph theory, algorithms, algebra, fixed point computations, etc



"The first reason Compiler Construction is such an important CS course is that it brings together, in a very concrete way, almost everything you learned before you took the course." Steve Yegge ♧

♧: Steve Yegge has worked at Amazon and Google, among others.

Compilers – A Microcosm of Computer Science♧

- **Algorithms**: graphs everywhere, union-find, dynamic programming
- **Artificial intelligence**: greedy algorithms, machine learning
- **Automata Theory**: DFAs for scanning, parser generators, context free grammars.
- **Algebra**: lattices, fixed point theory, Galois Connections, Type Systems
- **Architecture**: pipeline management, memory hierarchy, instruction sets
- **Optimization**: operational research, load balancing, packing, scheduling

♧:Shamelessly taken from slides by *Krishna Nandivada,* professor at the Indian Institute of Technology (http://www.cse.iitm.ac.in/~krishna/cs3300/lecture1.pdf)

Tainted Flow Analysis

```
$id = $_GET[”user”];

if ($id == '') {
  echo "Invalid user: $id"
} else {
  $getuser = $DB->query
      (”SELECT * FROM 'table' WHERE id='$id'”);
  echo $getuser;
}
```

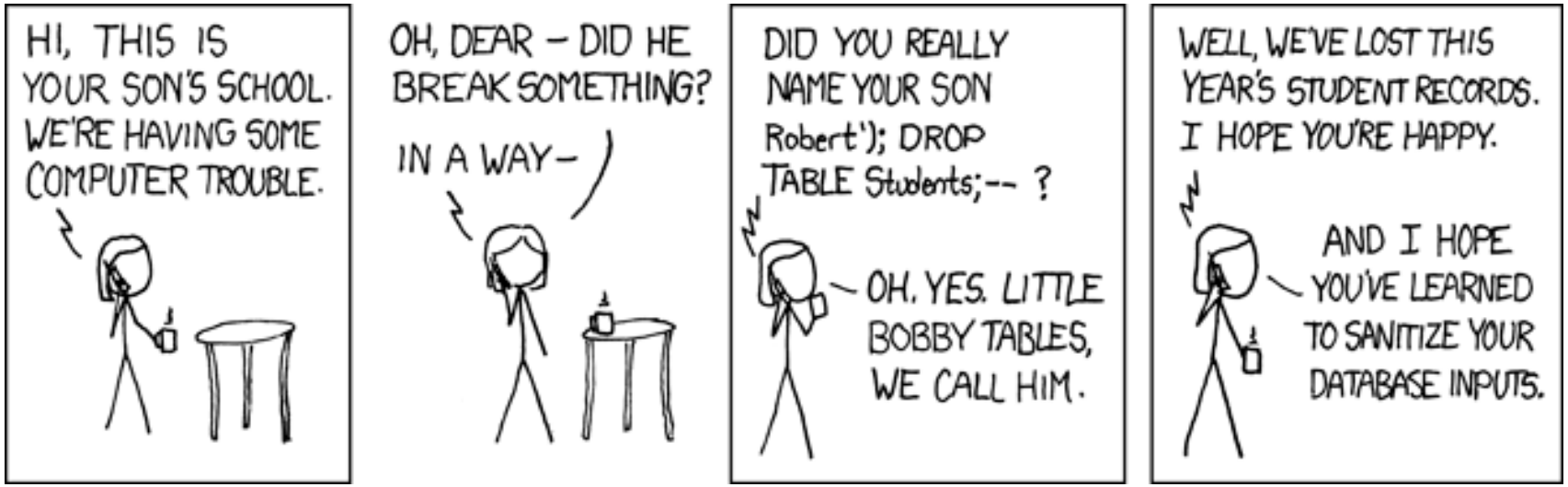
Does this program contain a security vulnerability?

There is a lot of cool things that compilers can do. The next slides contain a few examples of problems that compilers can solve today, and that maybe you would think that have nothing to do with compilation technology.

Tainted Flow Analysis

```
$id = $_GET[”user”];

if ($id == '') {
  echo "Invalid user: $id"
} else {
  $getuser = $DB->query
      (”SELECT * FROM 'table' WHERE id=’$id’”);
  echo $getuser;
}
```



Automatic Verification of Proofs

```
fun in_ord nil e = [e]
  | in_ord (h::t) e =
      if e < h then e::h::t
      else h :: in_ord t e

fun sort nil = nil
  | sort (h::t) = in_ord (sort t) h
```

1) Can you prove that this algorithm is correct?

2) What does it mean to prove that the algorithm is correct?

3) Can you use a tool to certify your proof?

Automatic Verification of Proofs

```
fun in_ord nil e = [e]
  | in_ord (h::t) e =
      if e < h then e::h::t
      else h :: in_ord t e

fun sort nil = nil
  | sort (h::t) = in_ord (sort t) h

fun sorted nil = true
  | sorted [e] = true
  | sorted (n1::n2::l) = n1 <= n2 andalso sorted (n2::l)
```

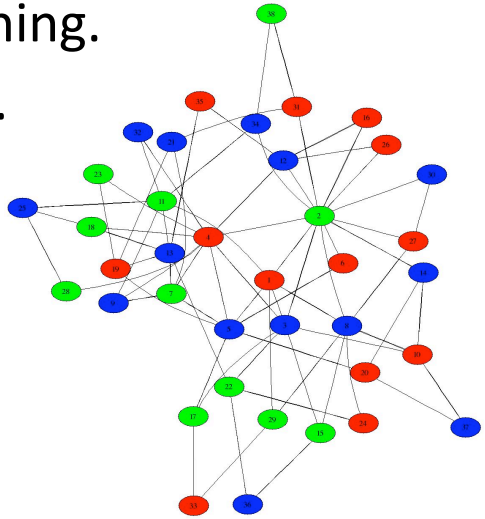
Prove that "`sorted (sort l) = true`" always

We need a formal specification of the problem that we need to solve. And we must prove some **invariant** that relates this specification and the algorithm that was designed to solve it.

Any idea on how types can helps us to prove something?

Register Allocation

- Sometimes, a new way to see things changes everything.
- Register allocation is one of the oldest optimizations.
  - It was already present in the earliest FORTRAN compilers.
- And for 50 years or so, people believed that the register assignment problem was NP-complete.
  - 50 years is a lot in computer science!



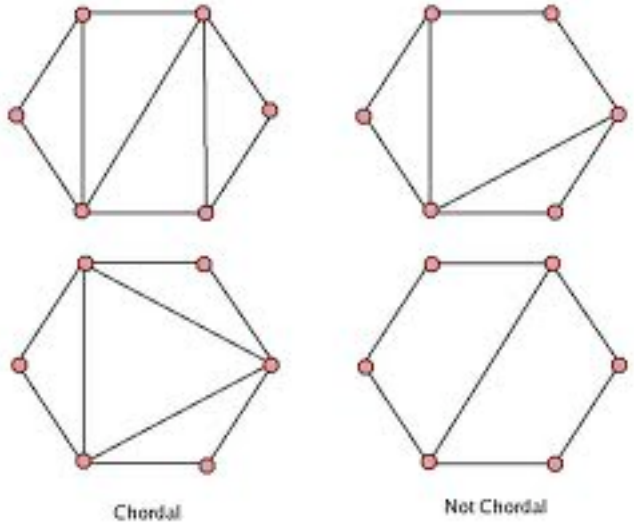
Given a program P, and a number of registers K, is there a valid way to assign the variables of P to the registers that does not use more than K registers?

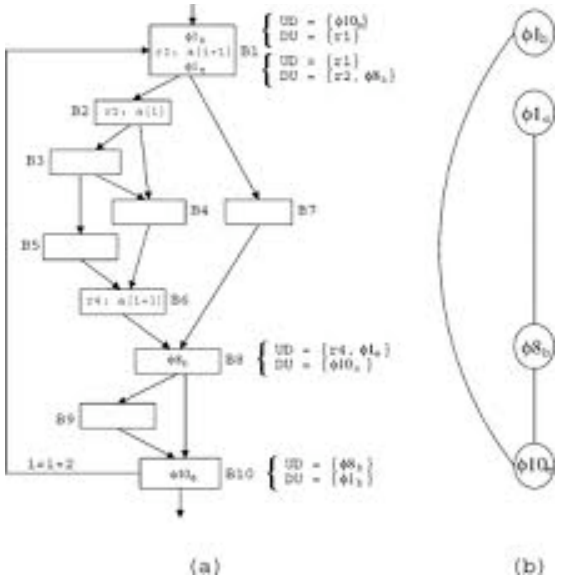
- But in 2005/06, the research community realized that this problem had, indeed, polynomial time solution.

Hold on: how can we solve this packing problem in polynomial time?

SSA-Based Register Allocation

- In 2005/06, different research groups realized that the interference graph of a program in Static Single Assignment (SSA) form is *chordal*♤.
- Chordal graphs can be colored exactly in polynomial time.
- Virtually any compiler uses this program representation today.
- The SSA form program never uses more registers than the original program.
  - It is a win-win situation.





♤: Register Allocation via the Coloring of Chordal Graphs, APLAS 2005

How Compiler Optimizations Work

- Usually, the compiler applies some static analysis to understand the code at hand, and then uses standard optimizations to improve this code.

```
#include <stdio.h>
int main() {
  int i = 0;
  int x = 2;
  int sum = 0;
  while (i++ < 100) {
    sum += x * x * x;
  }
  printf("%d\n", sum);
}
```

Nowadays, gcc has a number of arithmetic identities under its tool belt. Thus, the compiler can match this loop with an *arithmetic progression*. However, even if gcc did not know it, it could still optimize this program, removing the entire loop away. This optimization could be performed as the combination of two transformations: loop unrolling, and constant propagation.

What does this program do?

How Compiler Optimizations Work

- There are tradeoffs involved in many program transformations. If the compiler understands that the positive side of it shadows its negative side, then the optimization is performed.

```
#include <stdio.h>
int main() {
  int i = 0;
  int x = 2;
  int sum = 0;
  while (i++ < 100) {
    sum += x * x * x;
  }
  printf("%d\n", sum);
}
```

The compiler knows the number of iterations of the loop: 100. Thus, it can decide if it is worthwhile expanding the loop or not. There are many tradeoffs involved in program transformations: they can bring some benefits at the expense of some setbacks. For instance, if we unroll a very long loop, the program might grow too much, compromising cache locality. But in this case, 100 iterations of a small loop can be easily unrolled.

```
#include <stdio.h>
int main() {
  int i = 0;
  int x = 2;
  int sum = 0;
  sum += x * x * x;
  sum += x * x * x;
  ...
  sum += x * x * x;
  printf("%d\n", sum);
}
```

How Compiler Optimizations Work

- Compiler optimizations usually work together. We can infer that sum = 800 at the last line of our program, but we still want to remove the unnecessary computations.

```
#include <stdio.h>
int main() {
  int i = 0;
  int x = 2;
  int sum = 0;
  sum += x * x * x;
  sum += x * x * x;
  ...
  sum += x * x * x;
  printf("%d\n", sum);
}
```

Constant propagation is one of the most well-known program transformations. We know that sum = 0 at line three of our code. We propagate this information, thus inferring that sum = 8 right after, and then 16, 24, 32, ..., until we finally find that sum = 800 at the end of the program.

```
#include <stdio.h>
int main() {
  int i = 0;
  int x = 2;
  int sum = 0;
  sum += x * x * x;
  sum += x * x * x;
  ...
  sum += x * x * x;
  printf("%d\n", 800);
}
```

How Compiler Optimizations Work

- Some optimizations, such as constant folding, transform the program forwardly, from its beginning towards its end. Others, like dead code elimination, transform it backwards.

```
#include <stdio.h>
int main() {
  int i = 0;
  int x = 2;
  int sum = 0;
  sum += x * x * x;
  sum += x * x * x;
  ...
  sum += x * x * x;
  printf("%d\n", 800);
}
```

Variables that are defined, but not used anywhere are dead. The last definition of sum is dead, for instance. Once we remove an instruction that defines a dead variable, other variables might become dead as well. We keep iterating this process, until the program stops changing. At this point, we know that we are done. A important question is: will this algorithm eventually stabilize?

```
#include <stdio.h>
int main() {
  printf("%d\n", 800);
}
```

```
_main:
  pushl %ebp
  movl  %esp, %ebp
  pushl %ebx
  subl  $20, %esp
  call  L3
L3:
  popl  %ebx
  movl  $800, 4(%esp)
  movl  %eax, (%esp)
  call  _printf
  addl  $20, %esp
  popl  %ebx
  leave
  ret
```

THE ALLIES OF THE COMPILER WRITER

DCC 888

Static And Dynamic Analysis

- Compilers have two ways to understand programs:
  - Static Analysis
  - Dynamic Analysis
- Static analyses try to discover information about a program without running it.
- Dynamic analyses run the program, and collect information about the events that took place at runtime.

1) Can you give examples of dynamic analyses?

2) And can you give examples of static approaches?

3) What are the pros and cons of each approach?

Dynamic Analyses

- Dynamic analyses involve executing the program.
  - **Profiling**: we execute the program, and log the events that happened at runtime. Example: `gprof`.
  - **Test generation**: we try to generate tests that cover most of the program code, or that produce some event. Example: `Klee`.
  - **Emulation**: we execute the program in a virtual machine, that takes care of collecting and analyzing data. Example: `valgrind`.
  - **Instrumentation**: we augment the program with a meta-program, that monitors its behavior. Example: `AddressSanitizer`.

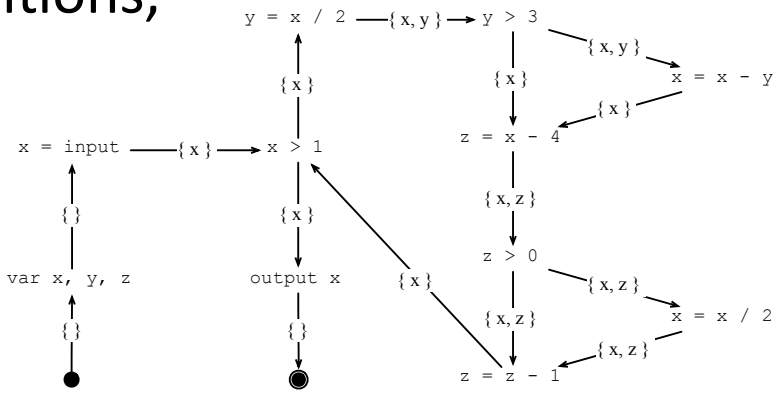
PPGCC
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO DCC-UFMG

Static Analyses

- In this course we will focus on static analyses.
- There are three main families of static analyses that we will be using:
  - **Dataflow analyses**: we propagate information based on the dependences between program elements, which are given by the syntax of the program.
  - **Constraint-Based analyses**: we derive constraints from the program. Relations between these constraints are not determined explicitly by the program syntax.
  - **Type analyses**: we propagate information as type annotations. This information lets us prove properties about the program, such as progress and preservation.

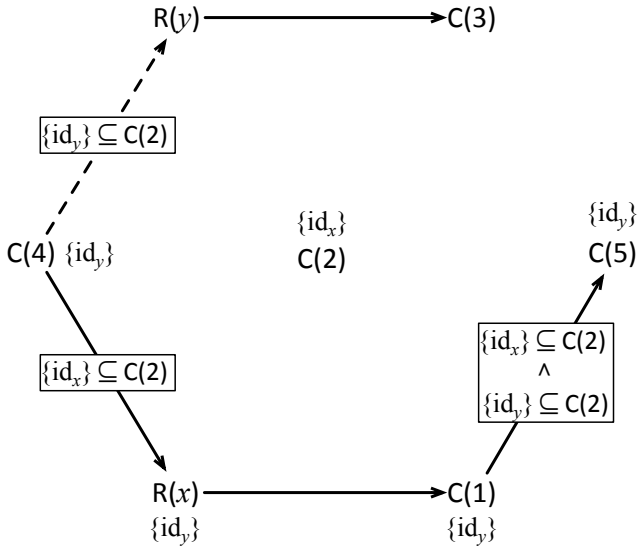
Dataflow Analyses

- Dataflow analyses discover facts about a program propagating information along the *control flow graph* of this program.
- The control flow graph of a program is a structure that makes explicit the directions along with the execution of the program might flow.
- Most of the analyses that we will see fit into this category:
  - Liveness analyses, reaching definitions, constant propagation, available expressions, very busy expressions, etc.



Constraint-Based Analyses

- Constraint-Based analyses are used when the directions in which we must propagate information cannot be easily discovered from the program's syntax.
- These analyses are usually handled as a set of constraints that we extract from the program code. These constraints have two forms:
  - lhs ⊆ rhs
  - $\{t\} \subseteq$ rhs' ⇒ lhs ⊆ rhs
- The two most important members of this family are control flow analysis, and pointer analysis.



Type Systems

- The dataflow analyses can usually be solved as type systems.
  - Types provide a good notation for the specification of these analyses.
  - They also simplify the proofs of correctness of the analyses, as they let us relate the static and dynamic semantics of a program.

$$[\textsc{Const}] \quad \forall \Gamma, \Gamma \vdash i : L, i = 0, 1$$

$$[\textsc{Secret}] \quad \forall \Gamma, \Gamma \vdash secret : H$$

$$[\textsc{Variable}] \quad \frac{\Gamma[x] = t}{\Gamma \vdash x : t}$$

$$[\textsc{Assign}] \quad \frac{\Gamma \vdash e : t}{pc \vdash \Gamma \ \{x = e\} \ \Gamma[x \mapsto pc \sqcup t]}$$

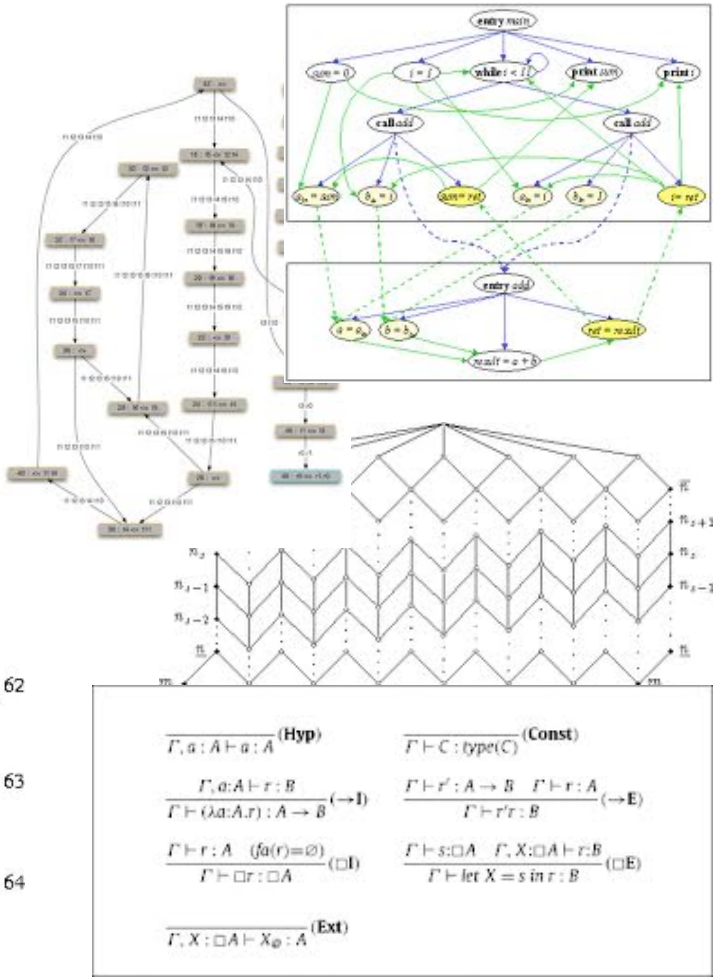
$$[\textsc{Branch}] \quad \frac{\Gamma \vdash e : t \qquad pc \sqcup t \vdash \Gamma\{c_i\}\Gamma', i = 1, 2}{pc \vdash \Gamma \ \{if \ e \ then \ c_1 \ else \ c_2\} \ \Gamma'}$$

$$[\textsc{Output}] \quad \frac{\Gamma \vdash e : t \qquad pc \sqcup t = L}{pc \vdash \Gamma \ \{output(e)\} \ \Gamma}$$

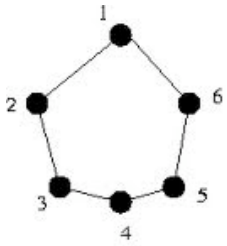
$$[\textsc{Chain}] \quad \frac{pc \vdash \Gamma\{c_1\}\Gamma' \qquad pc \vdash \Gamma'\{c_1\}\Gamma''}{pc \vdash \Gamma \ \{c_1; c_2\} \ \Gamma''}$$

The Broad Theory

- The algorithms used in compiler optimization include many different techniques of computer science
  - Graphs are everywhere
  - Lattices and the Fixed point theory
  - Many different types of induction
  - Dynamic programming techniques
  - Type theory
  - Integer Linear Programming
  - etc, etc, etc



And all of this in industrial strength compilers!



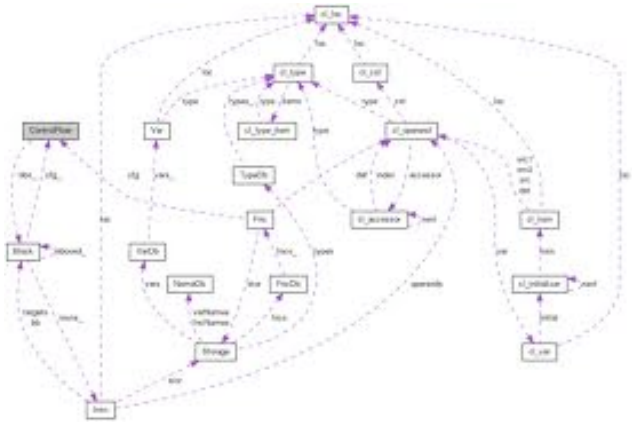
J-i = 2
13, 24, 35, 46, 51, 62

J-i = 3
14, 25, 36, 41, 52, 63

J-i = 4
15, 26, 31, 42, 53, 64
Finish with 16

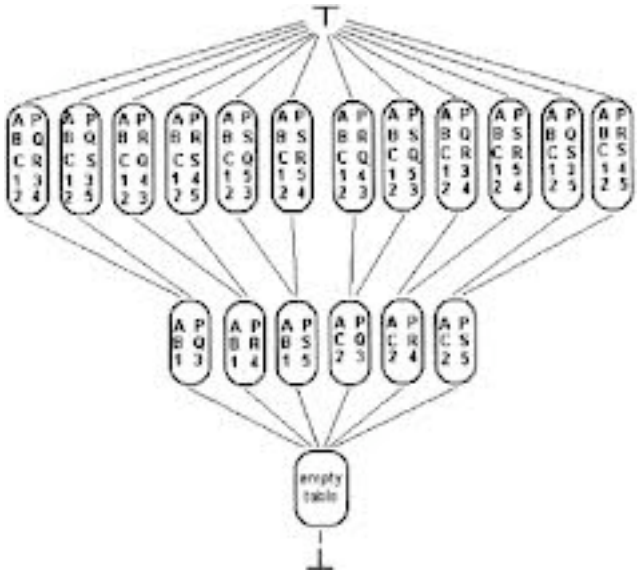
Graphs

- Graphs are the core of computer science, and permeate code optimization theory.
  - Graphs provide the basic program representation: the control flow graphs.
  - Graphs are in the center of most of the constraint solving systems.
  - Worklist algorithms usually rely on the dependence graph of a program to deliver faster results.
  - Strongly Connected Components and Topological Sorting are key to speedup dataflow analyses.
  - The most wide-spread register allocation technique relies on graph coloring.



Fixed Point Theory

- Most of the algorithms that we see in code optimization are iterative. A careless implementation may not terminate.
- These algorithms rely on finite lattices plus monotonic functions to ensure termination.
- If we always obtain more information after each iteration of our algorithm…
- and the total amount of information is finite…
- Then, eventually our algorithm must stabilize.



Induction all the way

- Most of our proofs are based on induction.
- We use three main types of induction:
  - Structural induction
  - Induction on derivation rules
  - Induction on size of syntactic terms

For instance: proving that a program terminates is undecidable in general; however, if the programming language is very simple, then we can show that their programs always terminate. To prove this, we must show that each evaluation of a term t gives us back a term t' that is smaller than t.

The main advantage of proofs by induction is that we already have technology to verify if these proofs are correct mechanically.

The Program Representation Zoo

- Depending on how we represent a program, we may facilitate different static analyses.
- Many dataflow analyses can be solved more efficiently in SSA form programs.
  - That is why this program representation is nowadays used in almost every compiler that is mildly important.
- As an example, the tainted flow problem used to be solved by cubic♧ algorithms until the late 2010's, when a faster, quadratic algorithm was proposed, based on a representation called e-SSA form♤.
- Minimal register assignment in SSA form programs has polynomial time solution, whereas it is NP-complete in general programs!

♤: Tainted Flow Analysis on e-SSA-Form Programs ♧: Trust in the lambda-Calculus

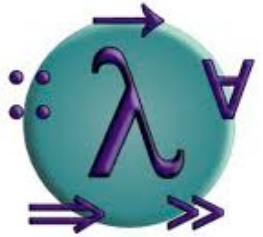
Open Source Community

- Many important compilers are currently open source.
- The Gcc toolkit has been, for many years, the most used compiler for C/C++ programs.

- LLVM is one of the most used compilers for research and in the industry.

- Mozilla's Monkey (SpiderMonkey, TraceMonkey, JagerMonkey, IonMonkey) family of compilers has a large community of users.

- Ocelot is used to optimize PTX, a program representation for graphics processing units.
- The Glasgow Haskell Compiler is widely used in functional programming research.



Conferences and Journals

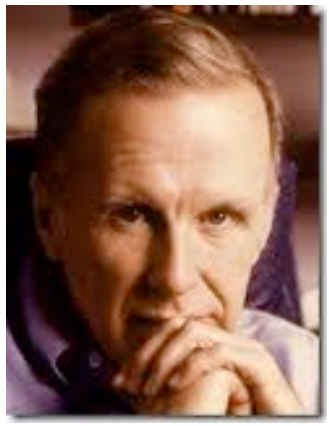
- There are many important conferences that accept compiler related papers:
  - PLDI: Programming Languages Design and Implementation (10.75, 18%)
  - POPL: Principles of Programming Languages (8.22, 20%)
  - ASPLOS: Architectural Support for Programming Languages and Operating Systems (11.49, 15%)
  - CGO: Code Generation and Optimization (3.56, 31%)
  - CC: Compiler Construction (2.46, 25%)
- And there is a very important journal in this field: TOPLAS – ACM Transactions on Programming Languages and Systems.

A Brief History of Optimizing Compilers

DCC 888

The Dawn of the First Compilers

- A more serious effort to move the task of generating code away from programmers started in the 50's.
- Fortran was one of the first programming languages, still in use today, to be compiled by an optimizing compiler.
- The developers of Fortran's compiler had to deal with two main problems:
  - Parsing
  - Code optimization
- One of the first challenges in terms of code optimization was *register allocation*.



Early Code Optimizations

- Frances E. Allen, working alone or jointly with John Cocke, introduced many of the concepts for optimization:
  - Control flow graphs
  - Many dataflow analyses
  - A description of many different program optimizations
  - Interprocedural dataflow analyses
  - Worklist algorithms
- A lot of these inventions and discoveries have been made in the IBM labs.

The Dataflow Monotone Framework

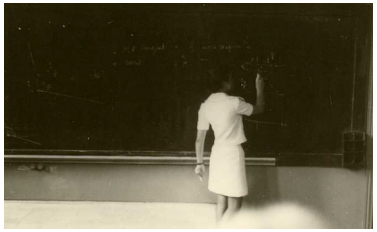
- Most of the compiler theory and technology in use today is based on the notion of the dataflow monotone framework.
  - Propagation of information
  - Iterative algorithms
  - Termination of fixed point computations
  - The meet over all paths solution to dataflow problems
- These ideas came, mostly, from the work of Gary Kildall, who is one of the fathers of the modern theory of code analysis and optimization.

The inventor of the BIOS system!

In addition of being paramount to the development of modern compiler theory, Gary Kildall used to host a talk show called "*The Computer Chronicles*". Nevertheless, he is mostly known for the deal with the Ms-DOS system that involved IBM and Bill Gates.

Abstract Interpretation

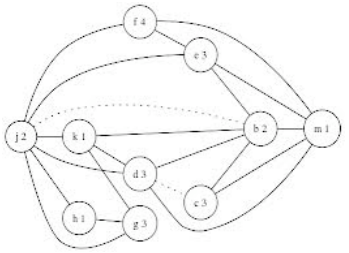
- Cousot and Cousot have published one of the most important papers in compiler research♤, giving origin to the technique that we call *Abstract Interpretation*.
- Abstract interpretation gives us information about the static behavior of a program.
- We could interpret a program to find out if a property about it is true.
  - But the program may not terminate, and even if it does, this approach could take too long.
- So, we assign abstract states to the variables, plus an operator called widening, that ensures that our interpretation terminates.
  - This approach may be conservative, but it does terminate!

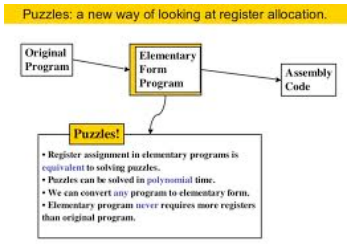


♤: Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints

Register Allocation

- Register allocation has been, since the early days of compilers, one of the most important code optimizations.
- Register allocation via graph-coloring was introduced by Gregory Chaitin in 1981.
- Linear scan, a register allocation algorithm normally used by JIT compilers, was introduced by Poletto and Sarkar in 1999.
- SSA-based register allocation was discovered independently by many researchers around 2005/06.





Static Single Assignment

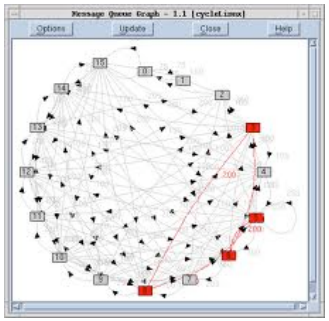
- Once in a while we see smart ideas. Perhaps, the smartest idea in compiler optimization was the Static Single Assignment Form.
  - A program representation in which every variable has only one definition site.
- SSA form was introduced by Cytron *et al*., in the late eighties♡ in IBM.
- There were many improvements since then, such as pruned SSA form, SSA-based register allocation, etc.
- The idea took off very quickly. Today almost every compiler uses this intermediate representation.

SSA Seminar: celebrated the 20th anniversary of the Static Single Assignment form, April 27-30, Autrans, France

♡: An Efficient Method of Computing Static Single Assignment Form

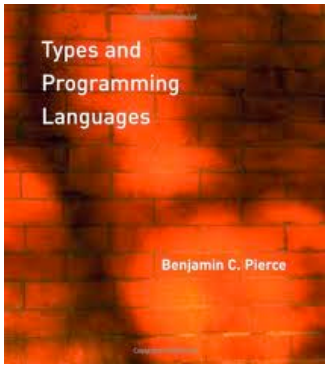
Constraint Based Analyses

- One of the most important elements of this family, the control flow analysis, was introduced by Olin Shivers in PLDI'88.
- Pointer analysis, another type of constraint based analysis, ubiquitous in current compilers, is the offspring of many fathers.
  - Lars O. Andersen launched the foundations of inclusion based-pointer analysis in 1994.
  - There has been a lot of work to speedup the algorithms. David Pearce, in 2003, explained the benefits of cycle detection.
  - In 1996, Bjarne Steensgaard described a less precise pointer analysis that could be solved in almost linear time.

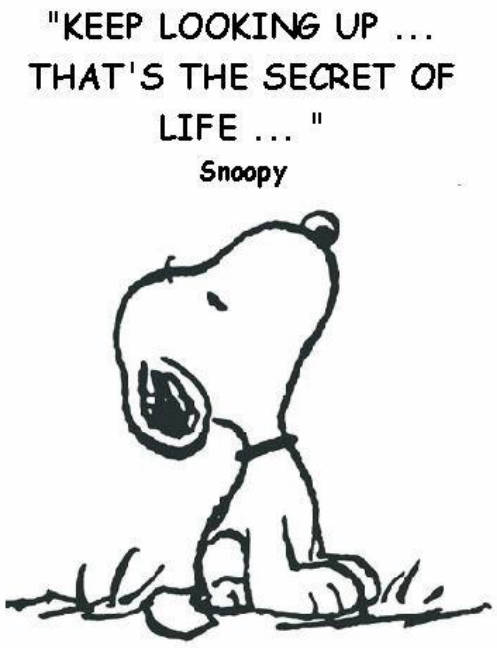


Type Theory

- Types have been around for a long time.
  - Philosophers and logicians would rely on types to solve paradoxes in Set Theory.
- A lot of work has been done by researchers in the functional programming field, such as Philip Wadler.
- Benjamin Pierce wrote a book, "Types and Programming Languages", in the early 2000's, that has been very influential in the field.
- A major boost in the mechanical verification of theorems is due to several independent groups, such as Xavier Leroy's, the father of Ocaml, and, Frank Pfenning's, with the Twelf system.



THE FUTURE



DCC 888

The Ever Increasing Gap

The Programming Languages are always evolving, usually towards higher levels of abstraction. And the hardware is also always evolving, towards greater performance.

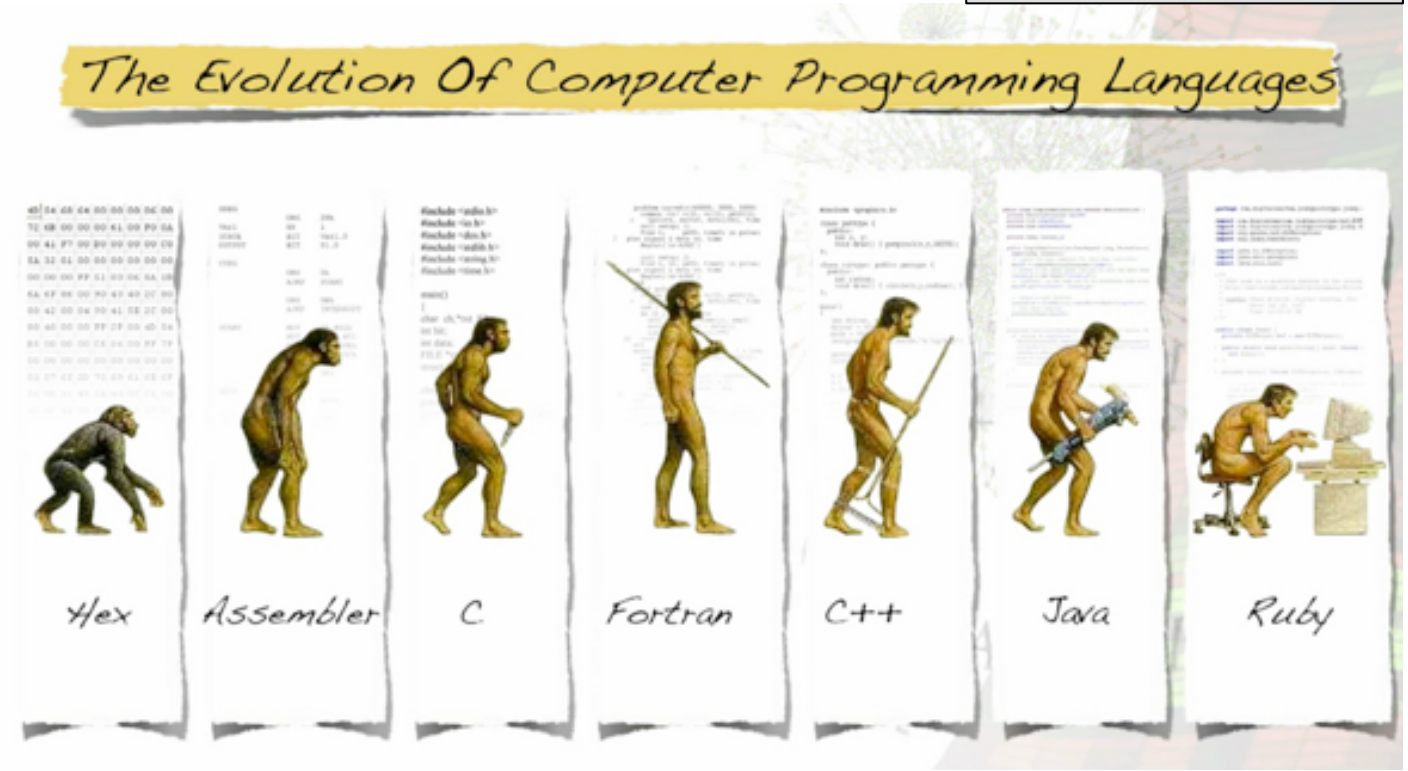
The compiler must bridge this ever increasing gap.

1) How the languages of the future will look like?

2) How the hardware of the future will look like?

3) Where is the research in compilers heading?

4) Is the compiler expert becoming more or less important?



The Future of Compiler Optimization

- The Full-Employment Theorem ensures that compiler writers will have a lot of work to do in the days ahead.
- Some influent researchers believe that research and development in the field will be directed towards two important paths, in the coming years♤:
  - Automatic parallelization of programs.
  - Automatic detection of bugs.
- Given that everything happens so fast in computer science, we are likely to see these new achievements coming!
  - And they promise to be super fun ☺

♤: Hall, Padua e Pingali "Compiler Research: The Next Fifty Years" *Communications of the ACM 2009*

The Complete Text (by Snoopy)

It Was A Dark And Stormy Night

Part I

It was a dark and stormy night. Suddenly, a shot rang out! A door slammed. The maid screamed.

Suddenly, a pirate ship appeared on the horizon!

While millions of people were starving, the king lived in luxury. Meanwhile, on a small farm in Kansas, a boy was growing up.

Part II

A light snow was falling, and the little girl with the tattered shawl had not sold a violet all day.

At that very moment, a young intern at City Hospital was making an important discovery. The mysterious patient in Room 213 had finally awakened. She moaned softly.

Could it be that she was the sister of the boy in Kansas who loved the girl with the tattered shawl who was the daughter of the maid who had escaped from the pirates?

The intern frowned.

"Stampede!" the foreman shouted, and forty thousand head of cattle thundered down on the tiny camp. The two men rolled on the ground grappling beneath the murderous hooves. A left and a right. A left. Another left and right. An uppercut to the jaw. The fight was over. And so the ranch was saved.

The young intern sat by himself in one corner of the coffee shop. he had learned about medicine, but more importantly, he had learned something about life.

THE END