Optimizing Compilers

Vineeth Kashyap

Department of Computer Science, UCSB

SIAM Algorithms Seminar, 2014

Warning!!

"Optimizing" compiler is a misnomer

My Background

- Worked on a compiler team for 2 years at National Instruments R&D
- Worked on multiple compilers during my PhD for languages we designed
- Will be working on “compiler-related” research at GrammaTech Research starting January 2015

Credit to xkcd.com

The Beginnings

[. . . .] Programmers wrote assembly

[1952] Grace Hopper wrote the first compiler

[1957] John W. Backus created an optimizing compiler

The Success of Fortran Optimizing Compiler

- Programs took hours instead of weeks to write
- Programs became portable!
- Decisively won the battle against assembly
- Space and military projects used it extensively

FizzBuzz

```
                        ; Init the registers
mov dx,03030h           ; For easier printing, the number is
                        ;kept in Binary Coded Decimal, in
----

                        ;the DX register.
mov ah,0Eh              ; 0Eh is the IBM PC interrupt 10h
                        ;function that does write text on
                        ;the screen in teletype mode.
mov bl,100d             ; BL is the counter (100 numbers).
xor cx,cx               ; CX is a counter that will be used
                        ;for screen printing.
xor bh,bh               ; BH is the counter for counting
                        ;multiples of three.

writeloop:              ; Increment the BCD number in DX.
inc dl                  ; Increment the low digit
cmp dl,3Ah              ; If it does not overflow nine,
jnz writeloop1          ;continue with the program,
mov dl,30h              ;otherwise reset it to zero and
inc dh                  ;increment the high digit
writeloop1:
inc bh                  ; Increment the BH counter.
cmp bh,03h              ; If it reached three, we did
                        ;increment the number three times
                        ;from the last time the number was
                        ;a multiple of three, so the number
                        ;is now a multiple of three now,
jz writefizz            ;then we need to write "fizz" on the
                        ;screen.
cmp dl,30h              ; The number isn't a multiple of
jz writebuzz            ;three, so we check if it's a
cmp dl,35h              ;multiple of five. If it is, we
jz writebuzz            ;need to write "buzz". The program
                        ;checks if the last digit is zero or
                        ;five.
mov al,dh               ; If we're here, there's no need to
int 10h                 ;write neither "fizz" nor "buzz", so
mov al,dl               ;the program writes the BCD number
int 10h                 ;in DX
writespace:
mov al,020h             ;and a white space.
int 10h
dec bl                  ; Loop if we didn't process 100
jnz writeloop           ;numbers.

programend:             ; When we did reach 100 numbers,
cli                     ;the program flow falls here, where
hlt                     ;interrupts are cleared and the
jmp programend          ;program is stopped.

writefizz:              ; There's need to write "fizz":
mov si,offset fizz      ; SI points to the "fizz" string,
call write              ;that is written on the screen.
xor bh,bh               ; BH, the counter for computing the
                        ;multiples of three, is cleared.
cmp dl,30h              ; We did write "fizz", but, if the
jz writebuzz            ;number is a multiple of five, we
cmp dl,35h              ;could need to write "buzz" also:
jnz writespace          ;check if the number is multiple of
                        ;five. If not, write a space and
                        ;return to the main loop.
writebuzz:              ; (The above code falls here if
                        ;the last digit is five, otherwise
                        ;it jumps)
mov si,offset buzz      ;SI points to the "buzz" string,
call write              ;that is written on the screen.
jmp writespace          ; Write a space to return to the main
                        ;loop.

write:                  ; Write subroutine:
mov cl,04h              ; Set CX to the lenght of the string:
                        ;both strings are 4 bytes long.
write1:
mov al,[si]             ; Load the character to write in AL.
inc si                  ; Increment the counter SI.
int 10h                 ; Call interrupt 10h, function 0Eh to
                        ;write the character and advance the
                        ;text cursor (teletype mode)
loop write1             ; Decrement CX: if CX is not zero, do
ret                     ;loop, otherwise return from
                        ;subroutine.

fizz:                   ;The "fizz" string.
db "fizz"

buzz:                   ;The "buzz" string.
db "buzz"
```

Credit to Rosetta Code

FizzBuzz

```
                     ; Init the registers
mov dx,03030h        ; For easier printing, the number is
                     ;kept in Binary Coded Decimal, in
----

                     ;the DX register.
mov ah,0Eh           ; 0Eh is the IBM PC interrupt 10h
                     ;function that does write text on
                     ;the screen in teletype mode.
mov bl,100d          ; BL is the counter (100 numbers).
xor cx,cx            ; CX is a counter that will be used
                     ;for screen printing.
xor bh,bh            ; BH is the counter for counting
                     ;multiples of three.

writeloop:           ; Increment the BCD number in DX.
inc dl               ; Increment the low digit
cmp dl,3Ah           ; If it does not overflow nine,
jnz writeloop1       ;continue with the program,
mov dl,30h           ;otherwise reset it to zero and
inc dh               ;increment the high digit
writeloop1:
inc bh               ; Increment the BH counter.
cmp bh,03h           ; If it reached three, we did
                     ;increment the number three times
                     ;from the last time the number was
                     ;a multiple of three, so the number
                     ;is now a multiple of three now,
jz writefizz         ;then we need to write "fizz" on the
                     ;screen.
cmp dl,30h           ; The number isn't a multiple of
jz writebuzz         ;three, so we check if it's a
cmp dl,35h           ;multiple of five. If it is, we
jz writebuzz         ;need to write "buzz". The program
                     ;checks if the last digit is zero or
                     ;five.
mov al,dh            ; If we're here, there's no need to
int 10h              ;write neither "fizz" nor "buzz", so
mov al,dl            ;the program writes the BCD number
int 10h              ;in DX
writespace:
mov al,020h          ;and a white space.
int 10h
dec bl               ; Loop if we didn't process 100
jnz writeloop        ;numbers.

programend:          ; When we did reach 100 numbers,
cli                  ;the program flow falls here, where
hlt                  ;interrupts are cleared and the
jmp programend       ;program is stopped.

writefizz:           ; There's need to write "fizz":
mov si,offset fizz   ; SI points to the "fizz" string,
call write           ;that is written on the screen.
xor bh,bh            ; BH, the counter for computing the
                     ;multiples of three, is cleared.
cmp dl,30h           ; We did write "fizz", but, if the
jz writebuzz         ;number is a multiple of five, we
cmp dl,35h           ;could need to write "buzz" also:
jnz writespace       ;check if the number is multiple of
                     ;five. If not, write a space and
                     ;return to the main loop.
writebuzz:           ; (The above code falls here if
                     ;the last digit is five, otherwise
                     ;it jumps)
mov si,offset buzz   ;SI points to the "buzz" string,
call write           ;that is written on the screen.
jmp writespace       ; Write a space to return to the main
                     ;loop.

write:               ; Write subroutine:
mov cl,04h           ; Set CX to the lenght of the string:
                     ;both strings are 4 bytes long.
write1:
mov al,[si]          ; Load the character to write in AL.
inc si               ; Increment the counter SI.
int 10h              ; Call interrupt 10h, function 0Eh to
                     ;write the character and advance the
                     ;text cursor (teletype mode)
loop write1          ; Decrement CX: if CX is not zero, do
ret                  ;loop, otherwise return from
                     ;subroutine.

fizz:                ;The "fizz" string.
db "fizz"

buzz:                ;The "buzz" string.
db "buzz"
```

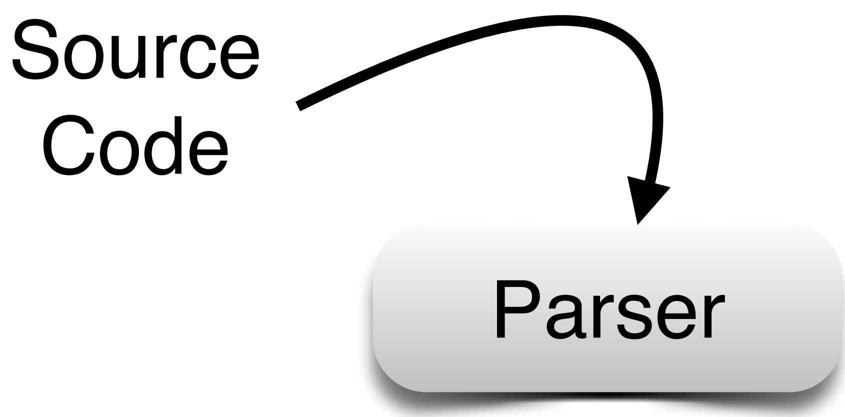
```
program fizzbuzz_if
  integer :: i

  do i = 1, 100
     if      (mod(i,15) == 0) then; print *, 'FizzBuzz'
     else if (mod(i,3) == 0) then; print *, 'Fizz'
     else if (mod(i,5) == 0) then; print *, 'Buzz'
     else;                        print *, i
     end if
  end do
end program fizzbuzz_if
```

Credit to Rosetta Code

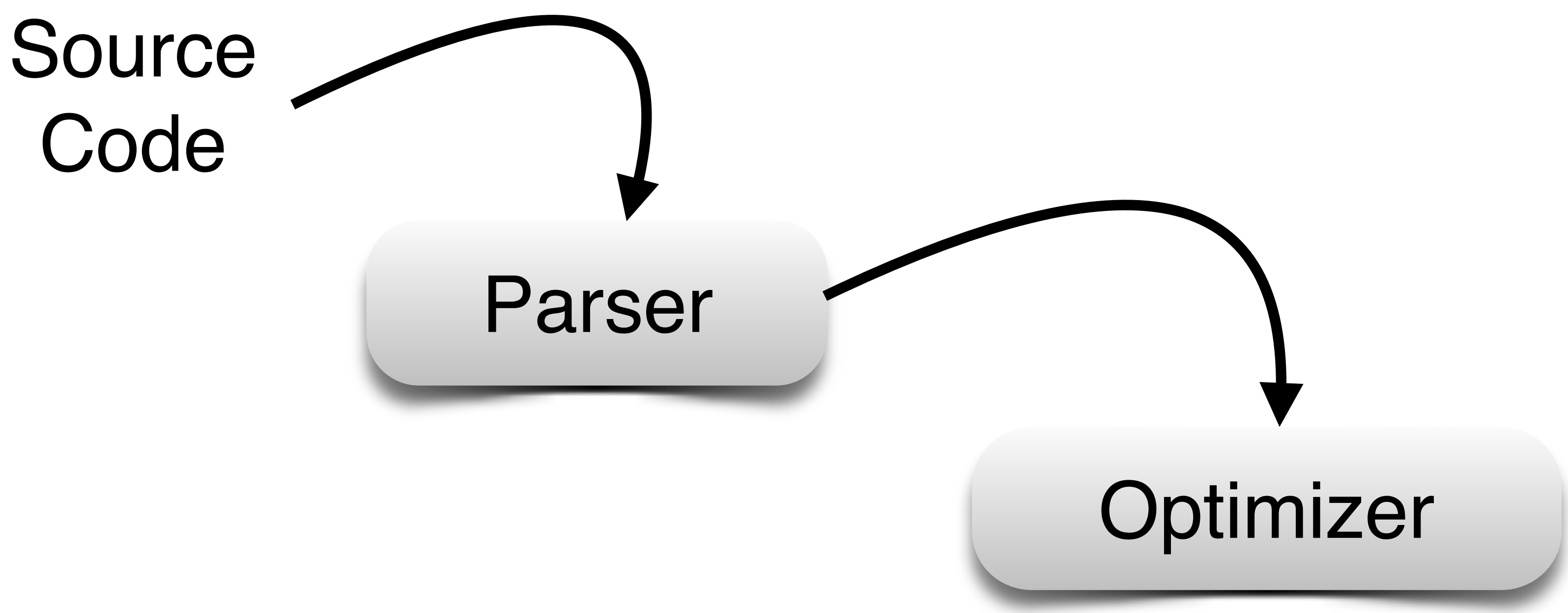
Compilers 101

Source
Code

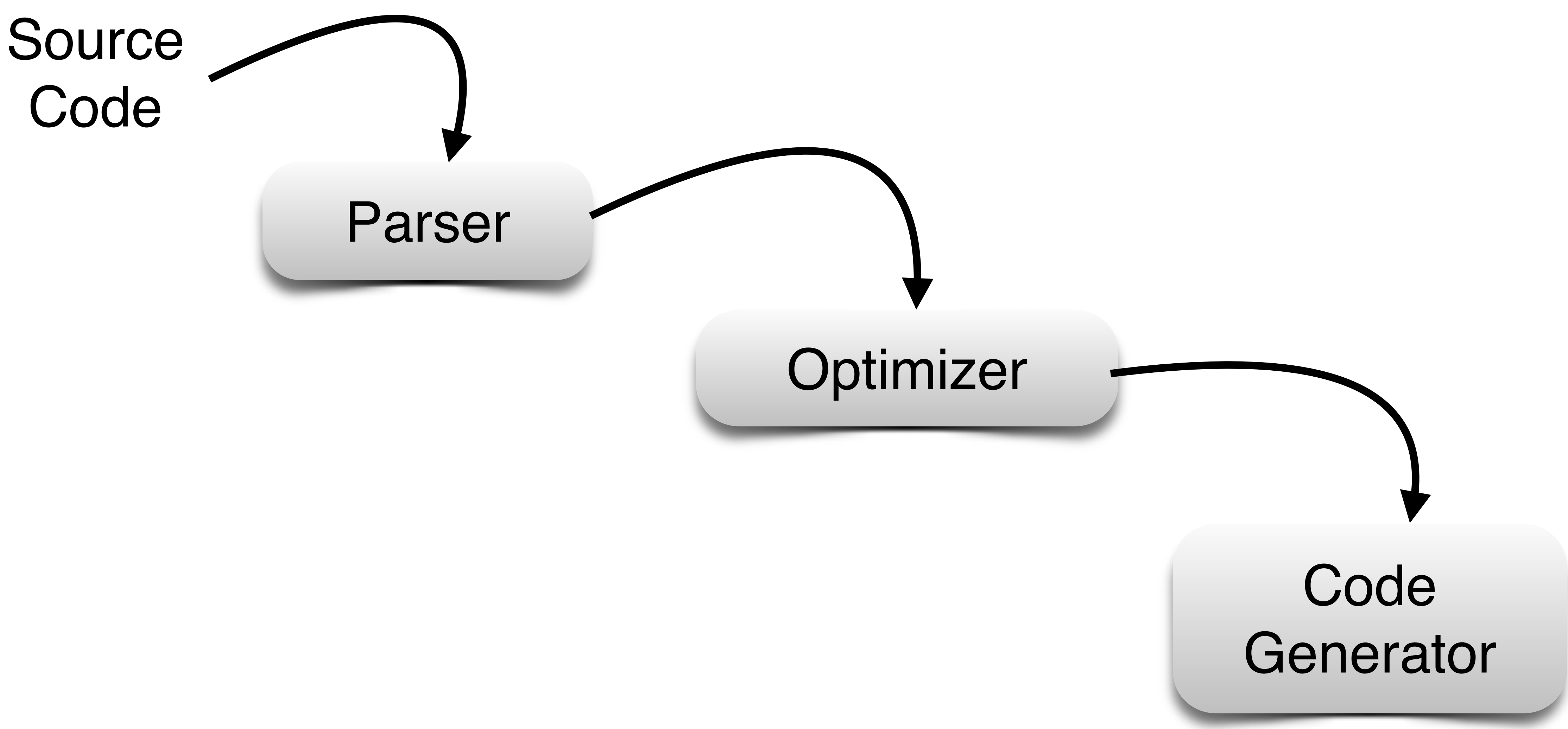
Compilers 101



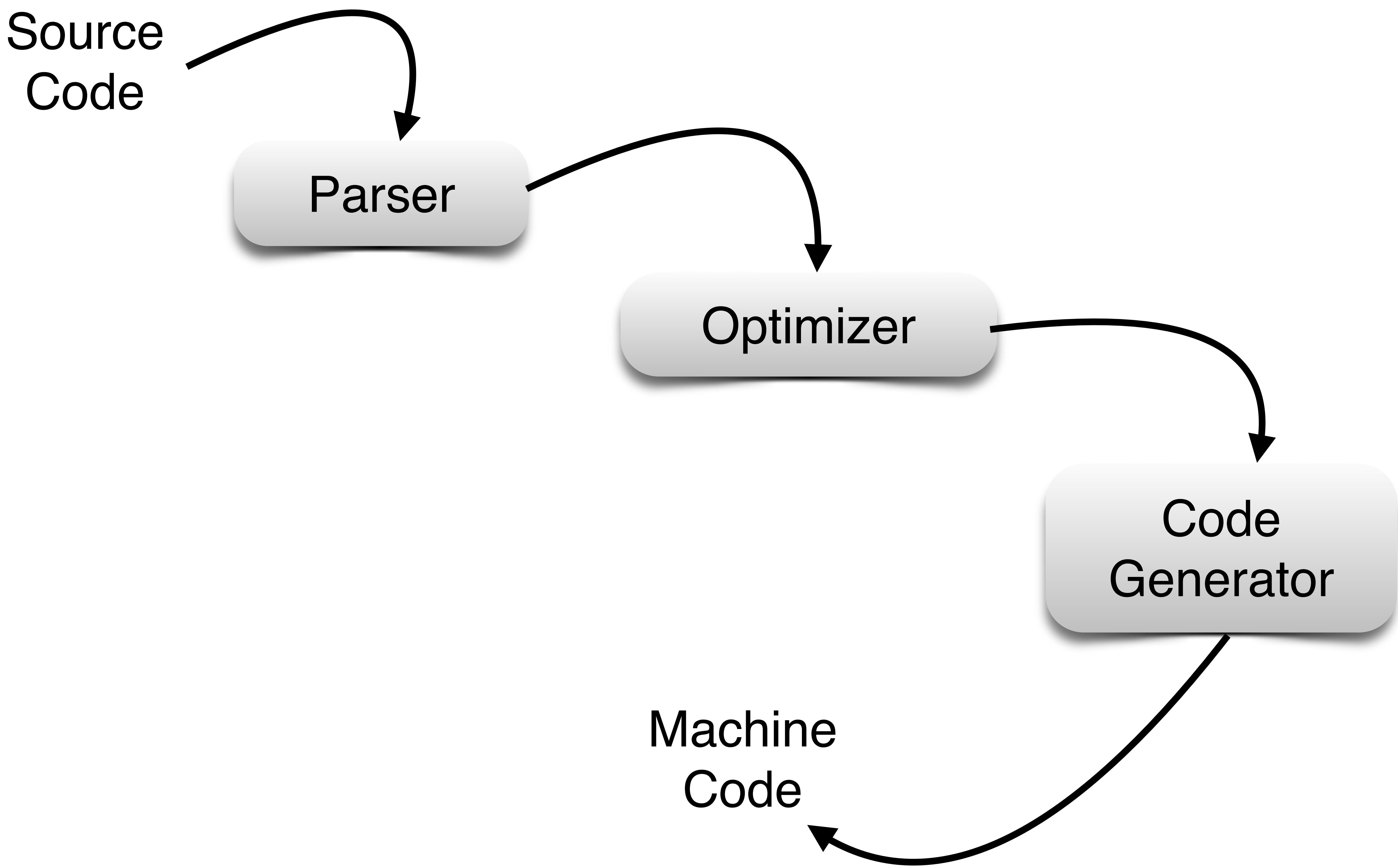
Compilers 101



Compilers 101



Compilers 101

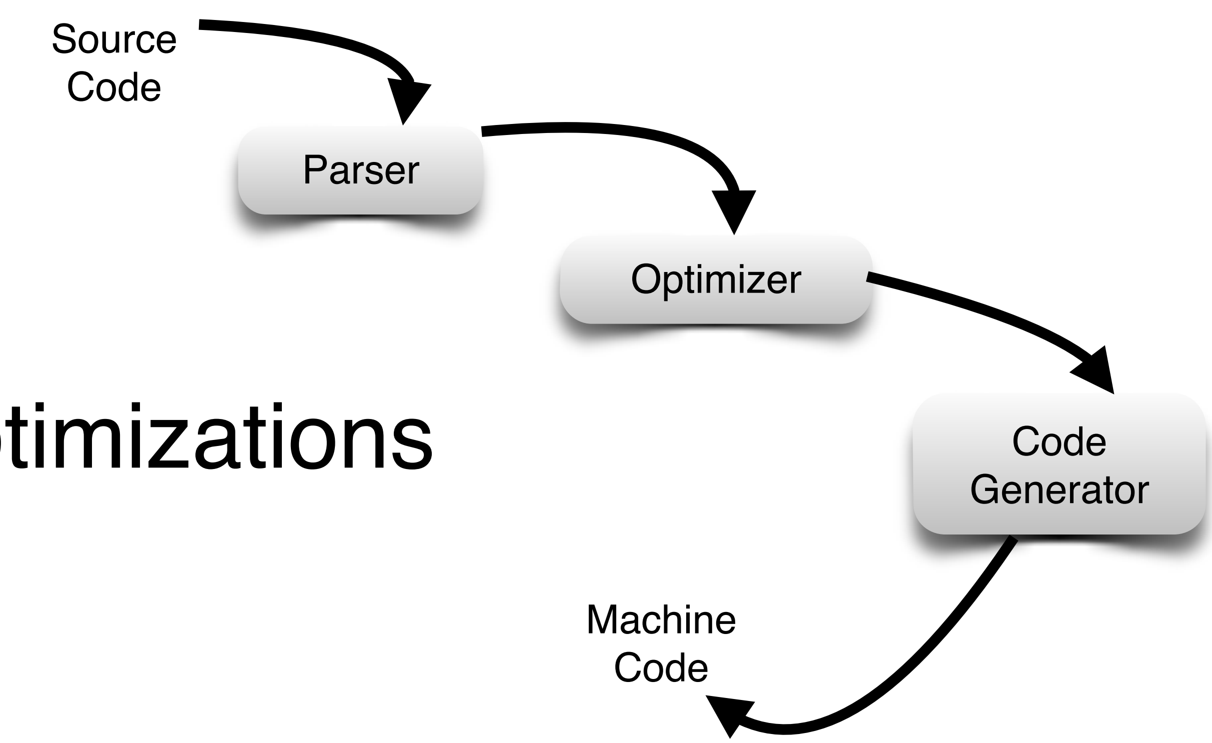


related.classes@cs.ucsb

- Compilers (CS 160)
- Programming Languages (CS 162)
- Program Analysis (CS 260)
- Modern Programming Languages (CS 263)

Rest of the Talk

- A brief bit on parsers
- Data flow analysis
- Some interesting high-level optimizations
- One low level optimization
- A couple of research ideas

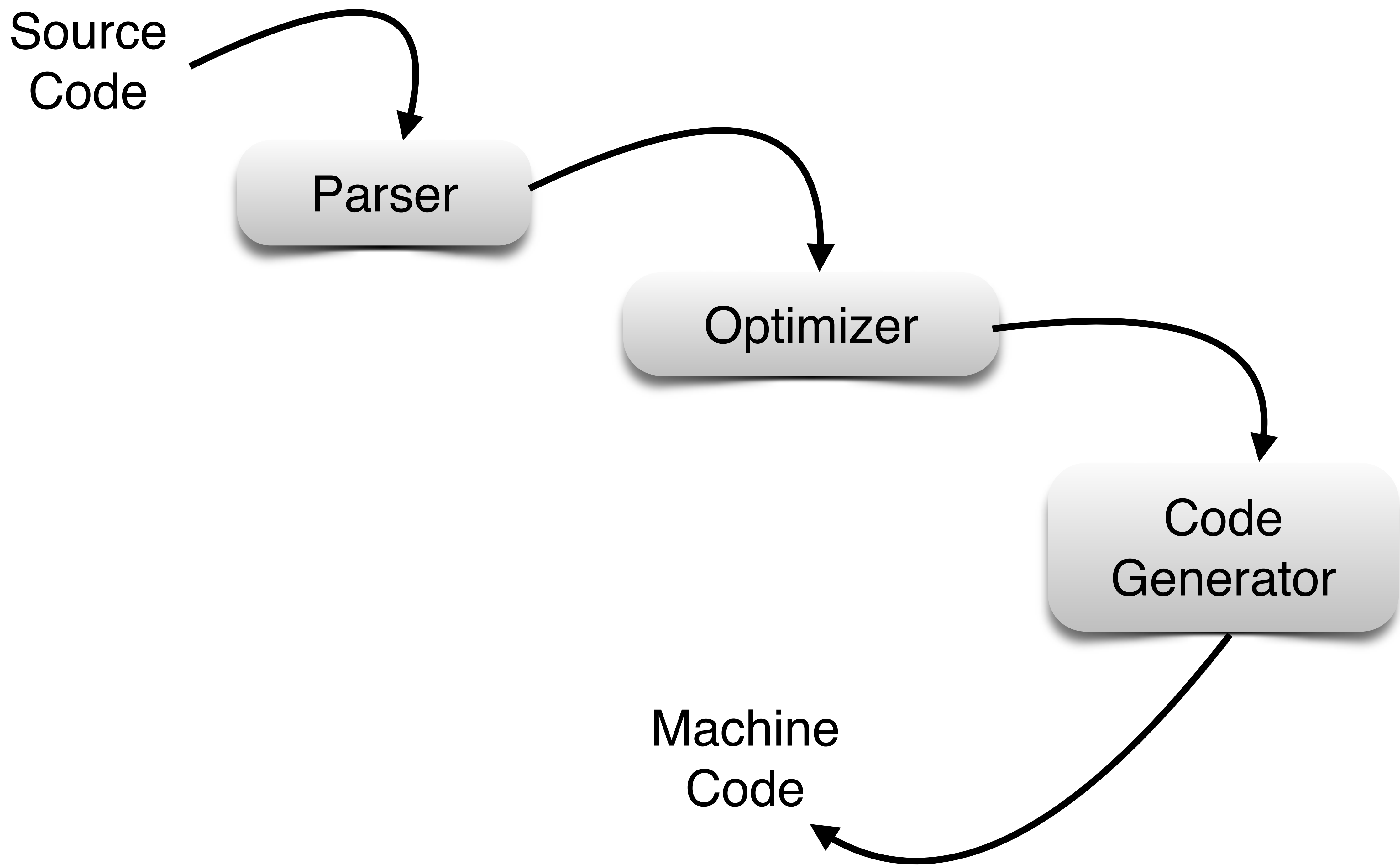


Parser

- Converts text (usually) to a structured representation
- Also recognizes a formal language
    - Regular languages (regular expressions)
    - Deterministic context-free grammars

HTML cannot be parsed using regular expressions

Regular expressions can only match regular languages but HTML is a context-free language


Source Code
Parser
Optimizer
Code Generator
Machine Code

Compiler Optimizations

- Finding *equivalent* programs that are *hopefully* better in some way
- It is undecidable what effect an optimization has on performance of a program
- It is undecidable whether an optimization is applicable!
- We opt for safety instead of missed optimization opportunity

Data Flow Analysis

- Figure out how a program manipulates its data
- Ex: “Does a variable always hold the same constant value at a given program point?”
- Ex: “Where are the given definition of a variable used?”
- Important enabler for a lot of optimizations

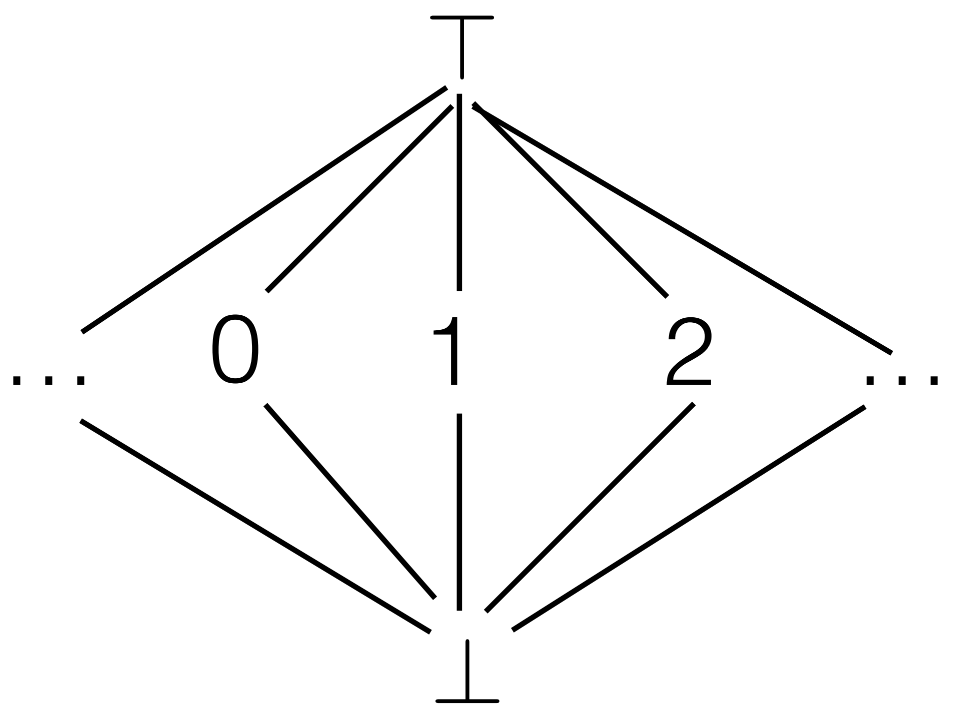
Conservative Analysis

- Over-approximation: can an optimization be safely performed? (no is acceptable, unless absolutely sure)
- False positives rather than false negatives
- Finds program invariants that hold across all possible executions of the program

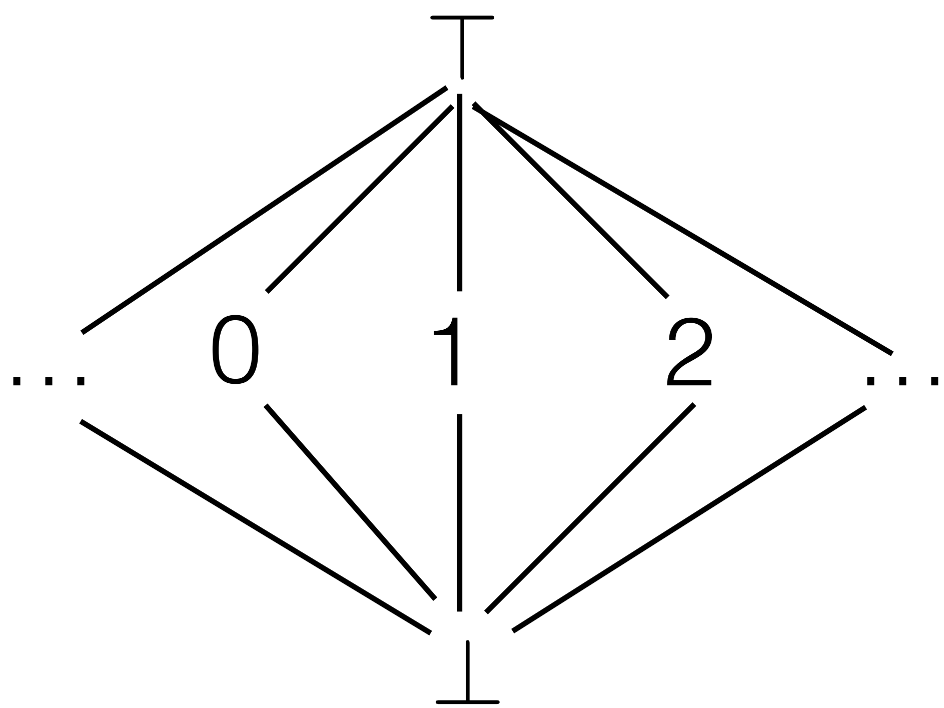
Data Flow Analysis Primer

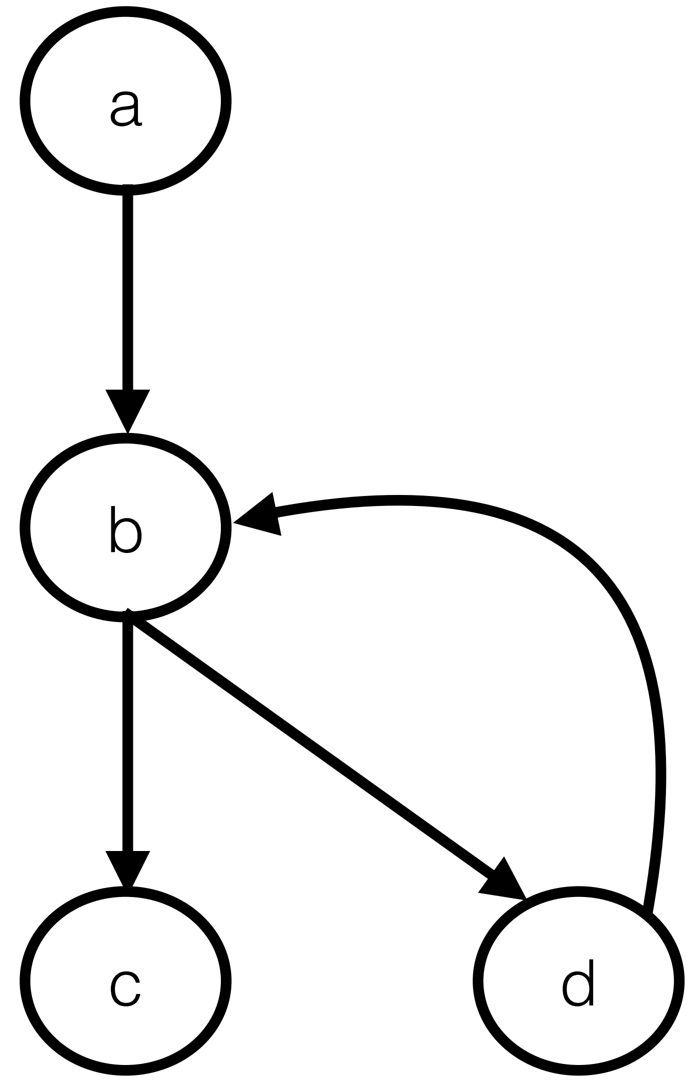
- Operates on elements of a lattice algebraic structure
- Solves simultaneous data flow equations
- Equations involve monotone computations over lattice elements
- Can be solved iteratively, guaranteed to terminate

Constant Propagation

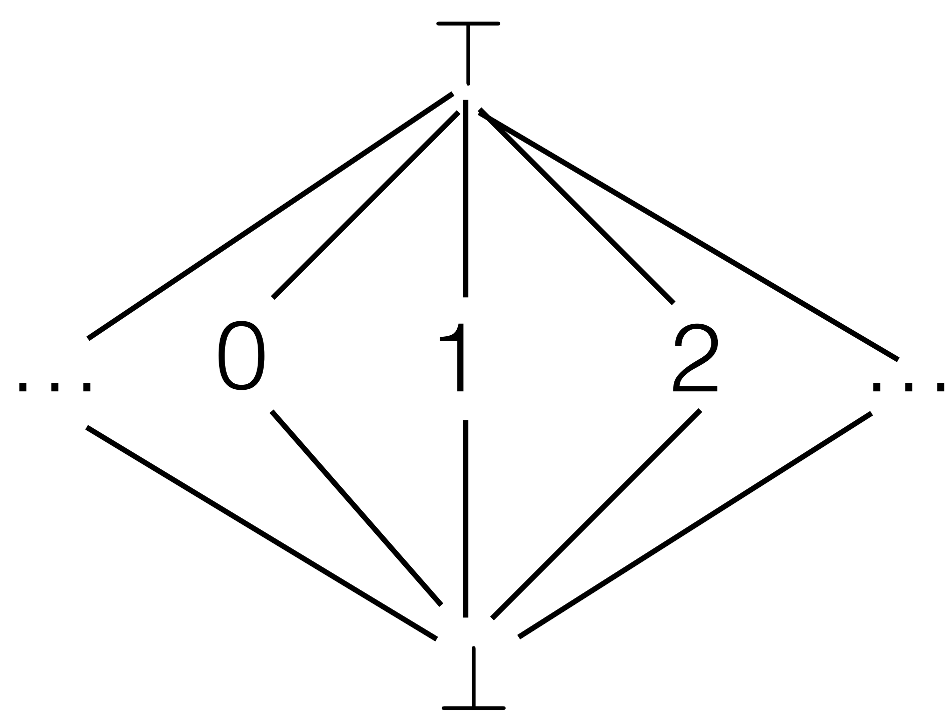


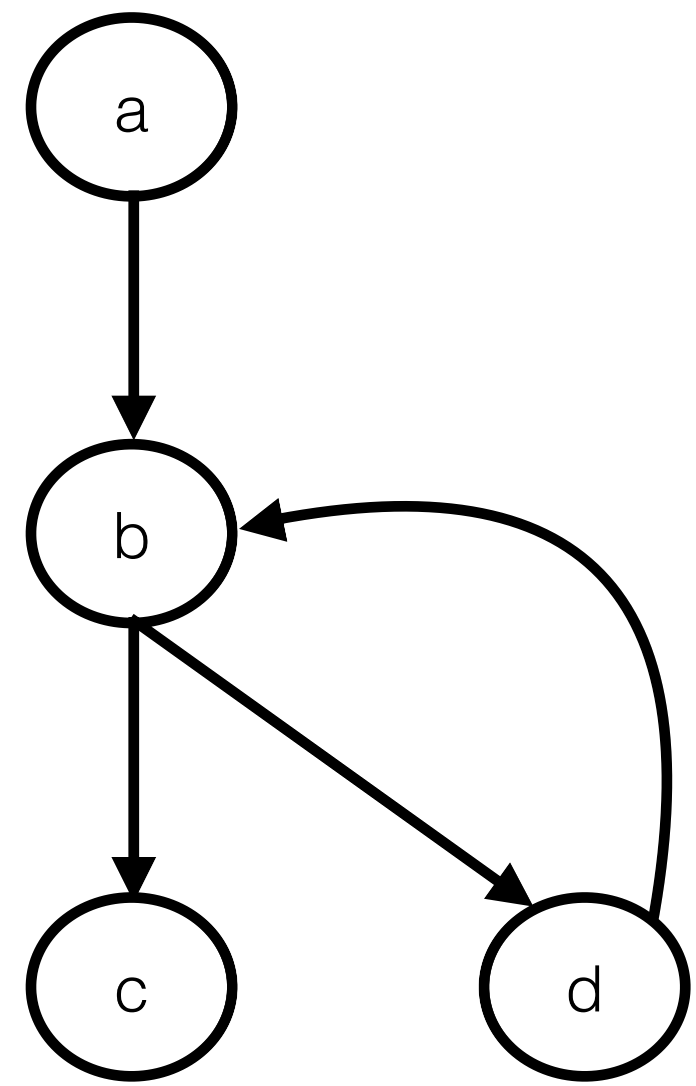
Constant Propagation





Constant Propagation





$$f_{out}(n) = f(T(n), \{f_{out}(k) \mid k \in pred(n)\})$$

Abstract Interpretation

- Can do whatever data flow analysis can do
- Is theoretically elegant, can provide proof of safety
- Mathematically relates the analysis with all executions
- What I use in my dissertation

Loop Invariant Code Motion

```
while (…) {
  constant := x/y;
  // other irrelevant code
}
```

Loop Invariant Code Motion

```
while (…) {
  constant := x/y;
  // other irrelevant code
}
```

```
constant := x/y;
while (…) {
 // other irrelevant code
}
```

Loop Invariant Code Motion

```
while (…) {
  constant := x/y;
  // other irrelevant code
}
```

```
if (…) constant := x/y;
while (…) {
 // other irrelevant code
}
```

Dead Code Elimination

```
var x := 0;
var y := 0;
if (x > 0) {
  // dead code
}
print x;
```

Tail Call Optimization

- Turn recursive calls into loops!
- Avoids costs associated with function calls and stack allocations

What are Tail Calls?

```
int sumToN(int n) {
  if (n == 1)
    return 1;
  else
    return n + sum(n-1);
} // sumToN(100);
```

What are Tail Calls?

```
int sumToN(int n) {
  if (n == 1)
    return 1;
  else
    return n + sum(n-1);
} // sumToN(100);
```

Not a tail call

What are Tail Calls?

```
int sumToN(int n) {
  if (n == 1)
    return 1;
  else
    return n + sum(n-1);
} // sumToN(100);
```

Not a tail call

```
int sumToN(int n, int acc) {
  if (n == 1)
    return acc;
  else
    return sum(n-1, acc+n);
} // sumToN(100, 1);
```

What are Tail Calls?

```
int sumToN(int n) {
  if (n == 1)
    return 1;
  else
    return n + sum(n-1);
} // sumToN(100);
```

Not a tail call

```
int sumToN(int n, int acc) {
  if (n == 1)
    return acc;
  else
    return sum(n-1, acc+n);
} // sumToN(100, 1);
```

Tail call

Tail Call Optimized Code

```
int sumToN(int n, int acc) {
  if (n == 1)
    return acc;
  else
    return sum(n-1, acc+n);
} // sumToN(100, 1);
```

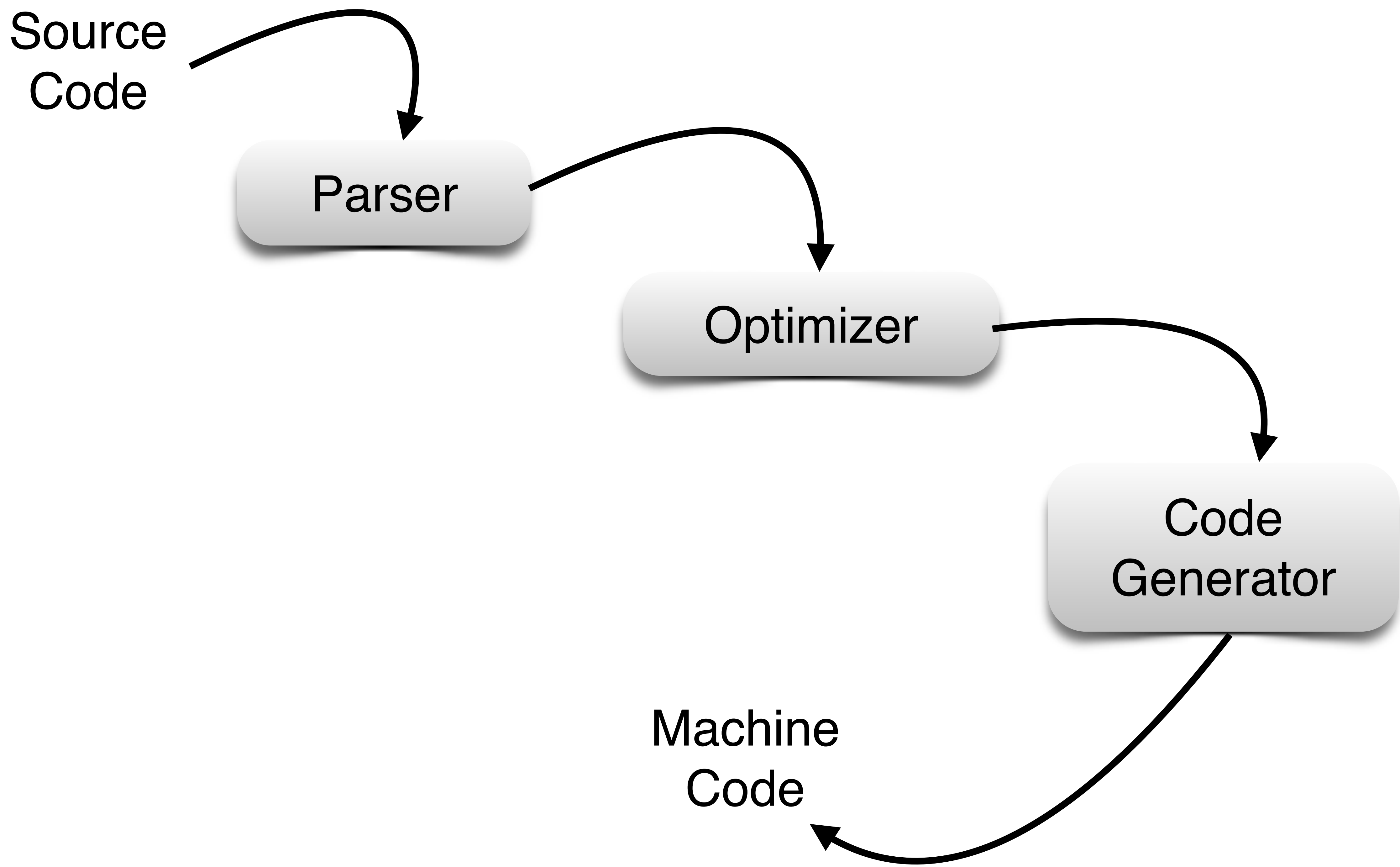
```
int sumToN(int n, int acc) {
 BEGIN:
   if (n == 1)
     return acc;
   else {
     acc = acc + n;
     n = n - 1;
     goto BEGIN;
   }
}
```

Tail Call Optimized Code

```
int sumToN(int n, int acc) {
  if (n == 1)
    return acc;
  else
    return sum(n-1, acc+n);
} // sumToN(100, 1);
```

```
int sumToN(int n, int acc) {
 BEGIN:
   if (n == 1)
     return acc;
   else {
     acc = acc + n;
     n = n - 1;
     goto BEGIN;
   }
}
```

Reassign Parameters


Source Code
Parser
Optimizer
Code Generator
Machine Code

Register Allocation

- Considered the most important optimization
- Minimize traffic between CPU registers and memory
- Usually requires a low level representation

What is Register Allocation

- Determine which values should be in which register
- Registers are scarce resource
- RISC: almost all operations are register based
- CISC: register operations are much faster

Note on Register Assignment

- RISC: usually trivial
- CISC: must take into account special registers

Using Graph Coloring!

- Register allocation can be viewed as graph-coloring [Cocke1971]
- Designed and implemented at IBM [Chaitin1981]

Example Allocation

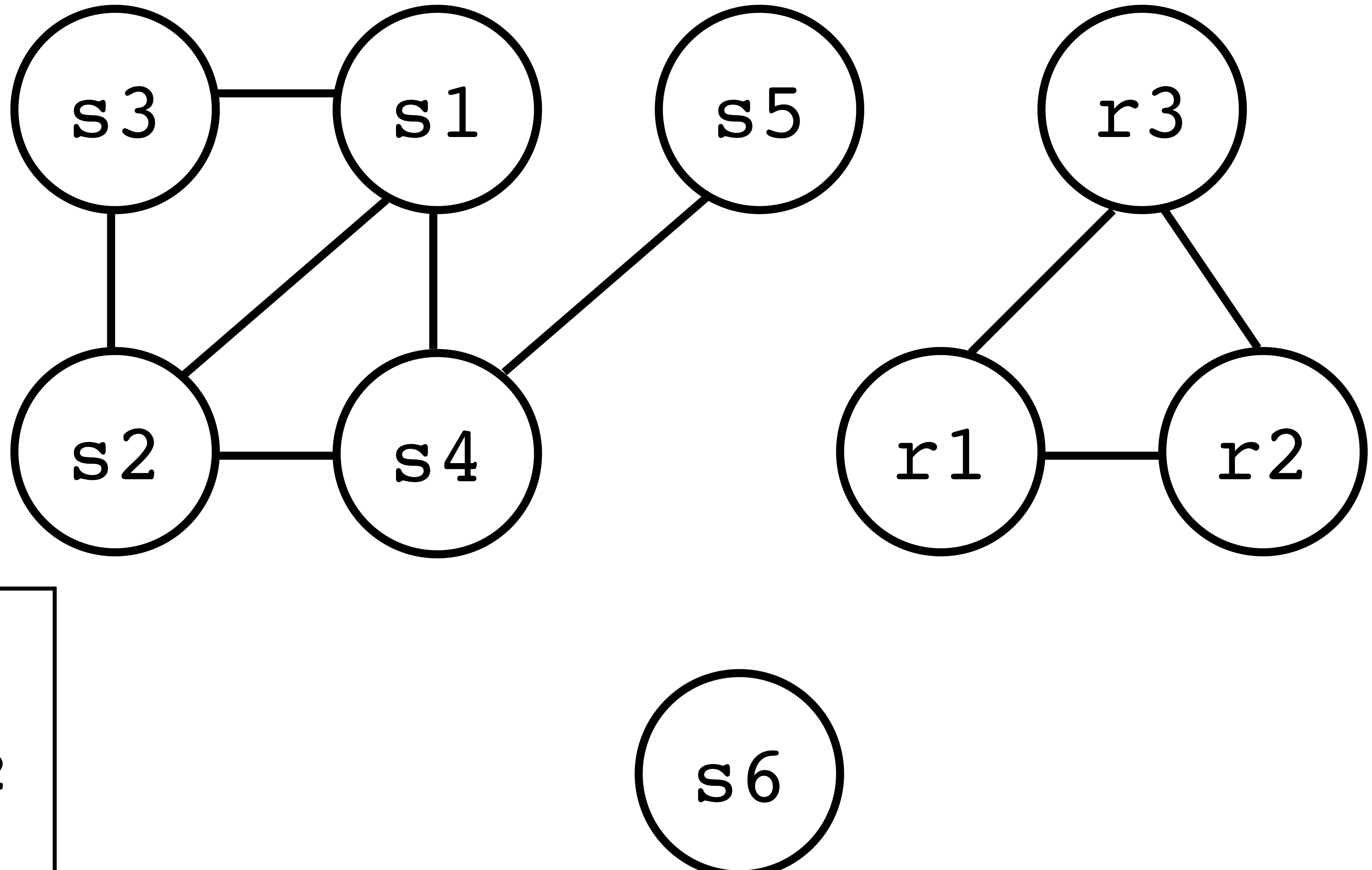
```
x := 2
y := 4
w := x + y
z := x + 1
u := x * y
x := z * 2
```

Symbolic Registers

```
x  := 2
y  := 4
w  := x + y
z  := x + 1
u  := x * y
x  := z * 2
```

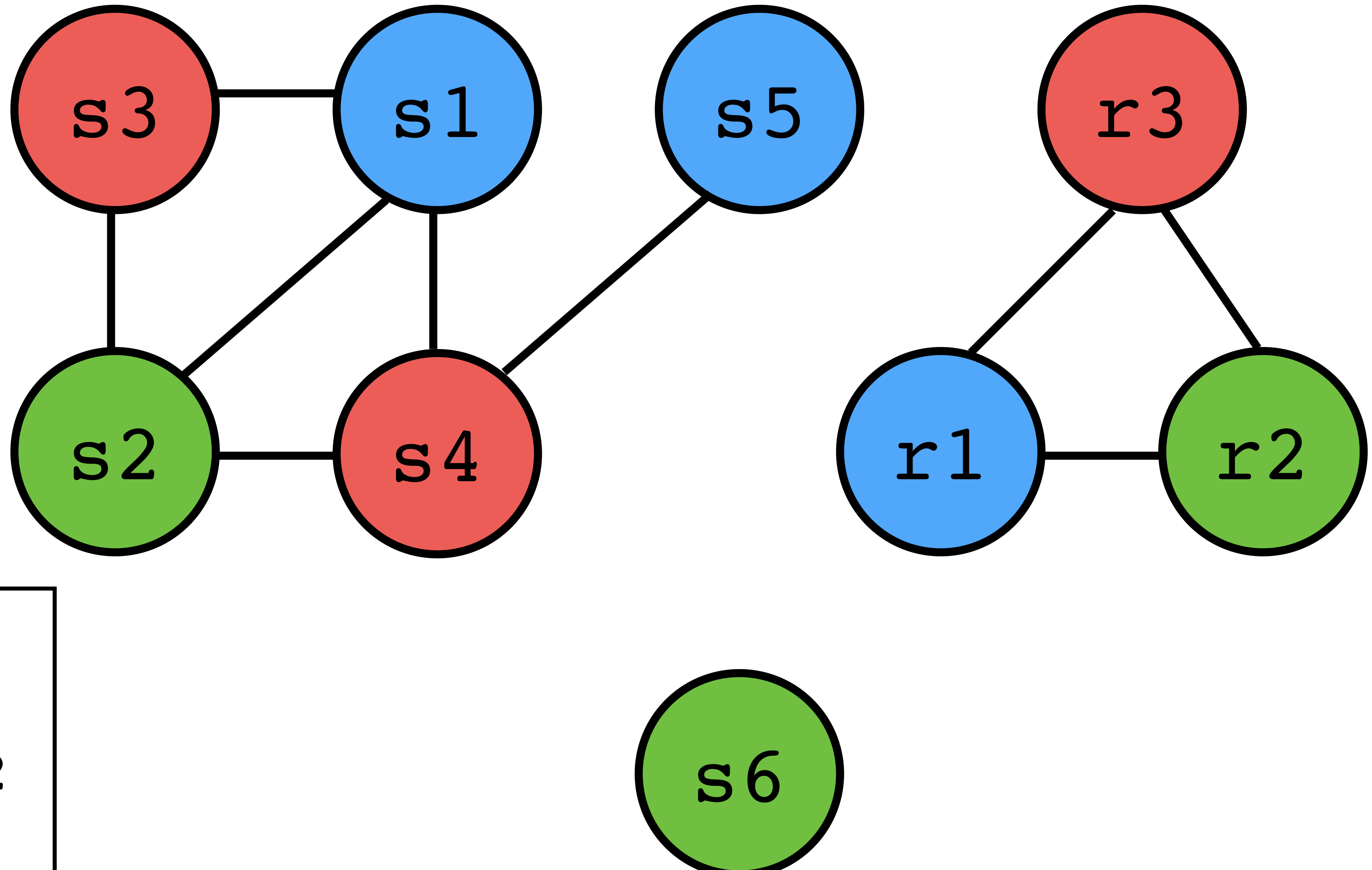
```
s1 := 2
s2 := 4
s3 := s1 + s2
s4 := s1 + 1
s5 := s1 * s2
s6 := s4 * 2
```

Interference Graphs




```
s1 := 2
s2 := 4
s3 := s1 + s2
s4 := s1 + 1
s5 := s1 * s2
s6 := s4 * 2
```

Graph Coloring

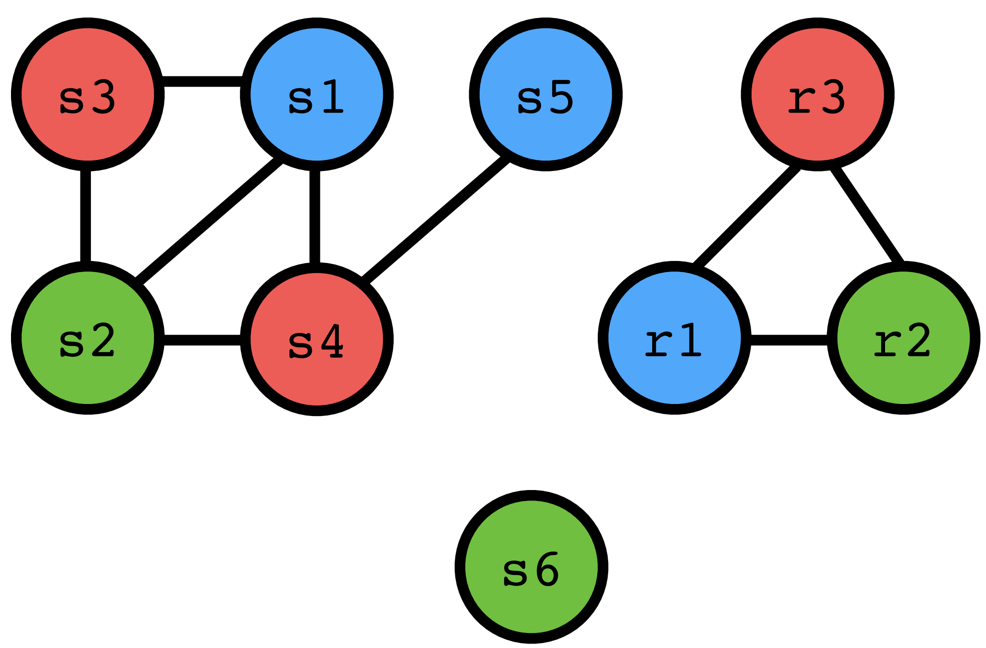



```
s1 := 2
s2 := 4
s3 := s1 + s2
s4 := s1 + 1
s5 := s1 * s2
s6 := s4 * 2
```

Allocate Registers

```
s1 := 2
s2 := 4
s3 := s1 + s2
s4 := s1 + 1
s5 := s1 * s2
s6 := s4 * 2
```

```
r1 := 2
r2 := 4
r3 := r1 + r2
r3 := r1 + 1
r1 := r1 * r2
r2 := r3 * 2
```



Register Spilling

What if the graph has a chromatic number k > number of registers?

Interesting Research Ideas

- Superoptimization
- Optimizations for reducing energy consumption

Advanced COMPILER DESIGN & IMPLEMENTATION

Steven S. Muchnick

Ping me!

Twitter: @vineethkashyap

Email: vineeth@cs.ucsb.edu

Website: http://www.cs.ucsb.edu/~vineeth/