Methodologies for Software Processes

Lecture 1

Grading

Seminar activity (about 2 assignments-implementation in Scala and an oral presentation of a paper) :-**-50%**

Final exam: **-- 50%**

- Final Written Exam (open books): **--50%**

Rules

- seminar activity will be done at the group level
- groups consist of max 2 students
- please form the groups and write your names in General/Files/Groups.xlsx
- final exam is individual and is an open book exam (you can have access at the lecture notes and the seminar notes)

References

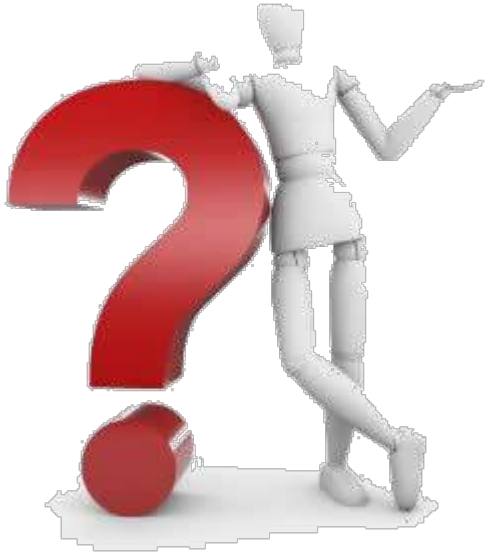
- Lecture notes and the draft book on Static Analysis, Anders Moller, Michael Schwartzbach, https://cs.au.dk/~amoeller/spa/
- F.Nielson, H.R.Nielson, C. Hankin: Principles of Program Analysis, Springer Verlag, 2004
- Other research papers that will be made available with the lecture notes

Note

Most of our slides are based on the Static Analysis course of Anders Moller and Michael Schwartzbach, from Aarhus University https://cs.au.dk/~amoeller/spa/from

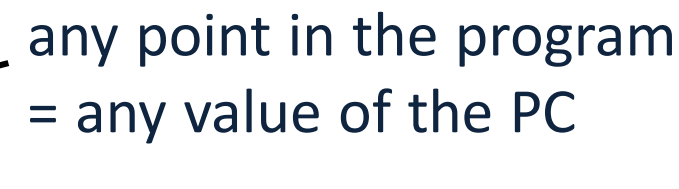
Questions about programs

- Does the program terminate on all inputs?
- How large can the heap become during execution?
- Can sensitive information leak to non-trusted users?
- Can non-trusted users affect sensitive information?
- Are buffer-overruns possible?
- Data races?
- SQL injections?
- XSS?
- …



Program points

```
foo(p,x) {
  var f,q;
  if (*p==0) { f=1; }
  else {
    q = malloc;
    *q = (*p)-1;
    f=(*p)*(x(q,x));
  }
  return f;
}
```



Invariants:

A property holds at a program point if it holds in any such state for any execution with any input

Questions about program points

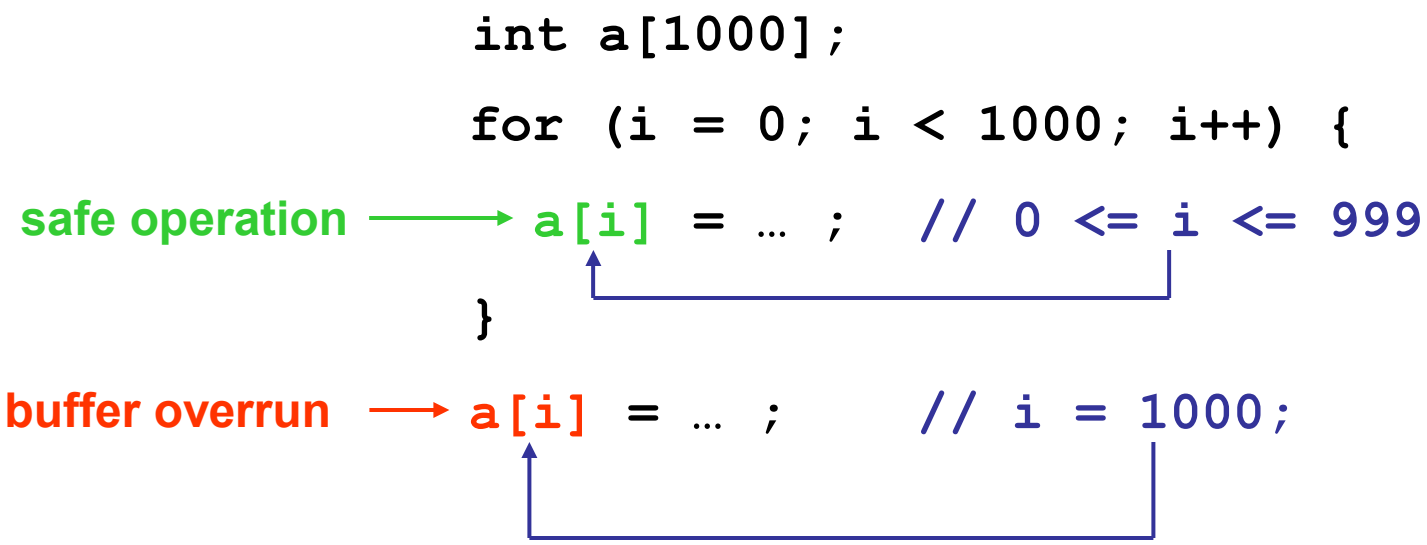
- Will the value of `x` be read in the future?
- Can the pointer `p` be `null`?
- Which variables can `p` point to?
- Is the variable `x` initialized before it is read?
- What is a lower and upper bound on the value of the integer variable `x`?
- At which program points could `x` be assigned its current value?
- Do `p` and `q` point to disjoint structures in the heap?
- Can this `assert` statement fail?

Why are the answers interesting?

- Increase efficiency
  - resource usage
  - compiler optimizations
  - …
- Ensure correctness
  - verify behavior
  - catch bugs early
  - …
- Support program understanding
- Enable refactorings

Program Verification

- Check that every operation of a program will never cause an error (division by zero, buffer overrun, deadlock, etc.)
- Example:



Why Is Software Verification Important?

- One of the most prominent challenges for IT.
  - Software bugs cost the U.S. economy about $59.5 billion each year (0.6% of the GDP) [NIST 02].
- Security is becoming a necessity.
  - The worldwide economic loss caused by all forms of overt attacks is $226 billion.
- Software defects make programming so painful.
- Stories
  - The Role of Software in Spacecraft Accidents(http://sunnyday.mit.edu/papers/jsr.pdf)
  - Hisory's Worst Software Bugs(http://www.wired.com/software/coolapps/news/2005/11/69355)

Static Program Analysis

- **Testing**: manually checking a property for some execution paths

- **Model checking**: automatically checking a property for all execution paths
- **Static analysis** consists of automatically discovering properties of a program that hold for all possible execution paths of the program

Static Analysis

Formal = based on rigorous mathematical logic concepts.

Semantics-based = based on a formal specification of the "meaning" of the program

Testing?

> *"Program testing can be used to show the presence of bugs, but never to show their absence."*
>
> [Dijkstra, 1972]

- Testing often takes 50% of development cost
- Errors in concurrent/distributed systems are hard to (re)produce with testing ("Heisenbugs")

Testing / Formal Verification

A very crude dichotomy:

Testing	Formal Verification
Correct with respect to the *set of test inputs*, and reference system	Correct with respect to *all inputs*, with respect to a *formal specification*
Easy to perform	Decidability problems, Computational problems,
Dynamic	Static

In practice:

Many types of testing,

Many types of formal verification.

Programs that reason about programs

Requirements to the perfect program analyzer

- SOUNDNESS (don’t miss any errors)
- COMPLETENESS (don’t raise false alarms)
- TERMINATION (always give an answer)

Rice's theorem, 1953

CLASSES OF RECURSIVELY ENUMERABLE SETS AND THEIR DECISION PROBLEMS(1)

BY

H. G. RICE

1. **Introduction.** In this paper we consider classes whose elements are recursively enumerable sets of non-negative integers. No discussion of recursively enumerable sets can avoid the use of such classes, so that it seems desirable to know some of their properties. We give our attention here to the properties of complete recursive enumerability and complete recursiveness (which may be intuitively interpreted as decidability). Perhaps our most interesting result (and the one which gives this paper its name) is the fact that no nontrivial class is completely recursive.

We assume familiarity with a paper of Kleene [5](2), and with ideas which are well summarized in the first sections of a paper of Post [7].

I. FUNDAMENTAL DEFINITIONS

2. **Partial recursive functions.** We shall characterize recursively enumer-

COROLLARY B. *There are no nontrivial c.r. classes by the strong definition.*

Rice's theorem

Any non-trivial property of the behavior of programs in a Turing-complete language is undecidable!

Reduction to the halting problem

- Can we decide if a variable has a constant value?

```
x = 17; if (TM(j)) x = 18;
```

- Here, x is constant if and only if the *j*'th Turing machine does not halt on empty input

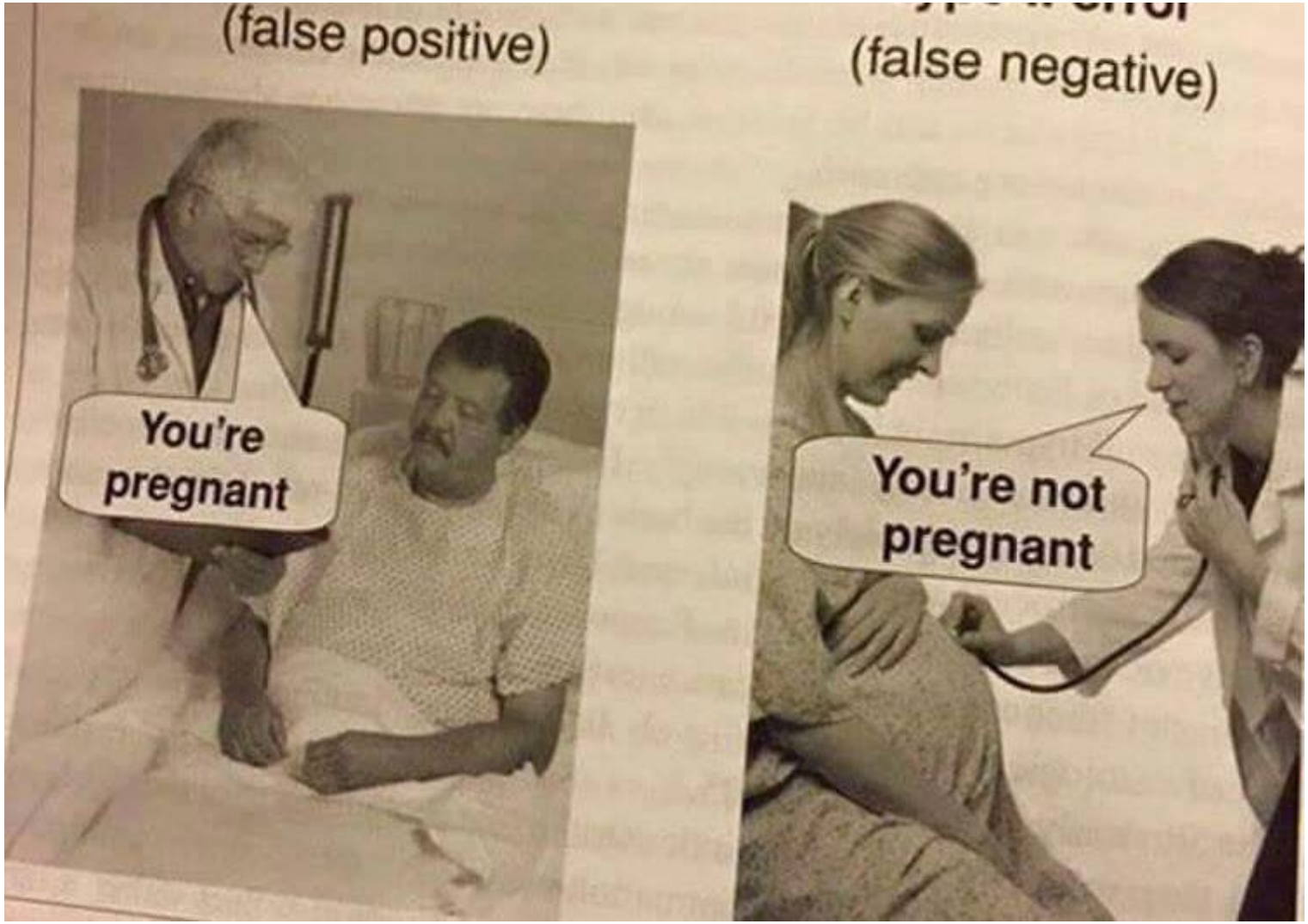
Approximation

- *Approximate* answers may be decidable!

- The approximation must be *conservative*:
  - i.e. only err on "the safe side"
  - which direction depends on the *client application*

- We'll focus on decision problems
- More subtle approximations if not only "*yes*"/"*no*"
  - e.g. memory usage, pointer targets

False positives and false negatives



Example approximations

- Decide if a given function is ever called at runtime:
  - if "*no*", remove the function from the code
  - if "*yes*", don't do anything
  - the "*no*" answer *must* always be correct if given

- Decide if a cast `(A)x` will always succeed:
  - if "*yes*", don't generate a runtime check
  - if "*no*", generate code for the cast
  - the "*yes*" answer *must* always be correct if given

Beyond “yes”/“no” problems

- How much memory / time may be used in any execution?

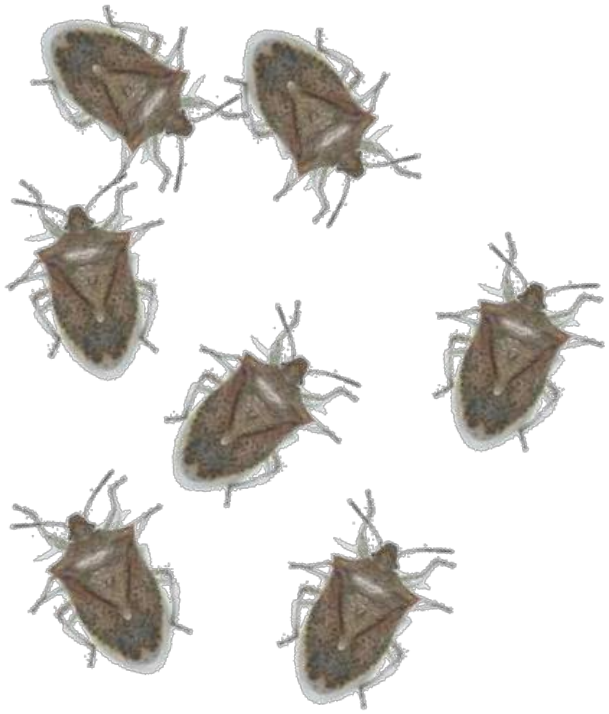
- Which variables may be the targets of a pointer variable p?

The engineering challenge

- A correct but trivial approximation algorithm may just give the useless answer every time
- The *engineering challenge* is to give the useful answer often enough to fuel the client application
- ... and to do so within reasonable time and space
- This is the hard (and fun) part of static analysis!

Bug finding

```
int main()
  { char *p,*q;
  p = NULL;
  printf("%s",p);
  q = (char *)malloc(100);
  p = q;
  free(q);
  *p = 'x';
  free(p);
  p = (char *)malloc(100);
  p = (char *)malloc(100);
  q = p;
  strcat(p,q);
}
```



```
gcc -Wall foo.c
lint foo.c

No errors!
```

Does a

focus
software devel

196 6 68

Uber Engineering

POSTED ON SEP 6, 2017 TO ANDROID, DEVELOPER TOOLS, IO

Finding inter-pr
Infer static anal


SAM BLACKSHEAR

DIN

The capabilities of static analyzers, w
our work on the Infer static analyzer,
source analysis tools like Findbugs,
procedural bugs, or bugs that involve

We'll take a look at two examples of
source DuckDuckGo Android app, a
the tools mentioned above, which pe
Analyzer — only intra-file analysis (p
unit, a file-with-includes).

Inter-procedural bugs are significant,
Facebook developers have fixed tho
can have a large impact; we include
Facebook. As we have found, inter-p
codebases that consist of millions of

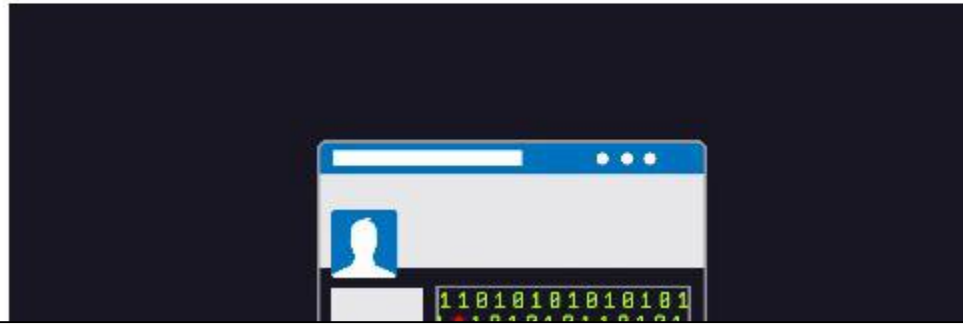
WIRED How Facebook Catches Bugs in Its 100 Million

BUSINESS CULTURE GEAR IDEAS

SHARE

LILY HAY NEWMAN SECURITY 08.15.19 05:03 PM

HOW FACEBOOK CATCHES BUGS IN ITS 100 MILLION LINES OF CODE

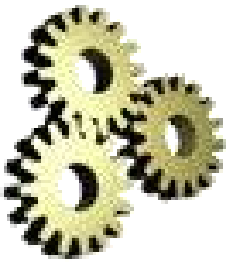


A constraint-based approach

Conceptually separates the analysis specification from algorithmic aspects and implementation details

```
public class Matrix {
public static void main(String[] args) {
        int arr[][]=new int[3][3];
        System.out.println("Enter nine elements");
        Scanner sc=new Scanner(System.in);
        for(int i=0;i<arr.length;i++)
        {
            for(int j=0;j<arr.length;j++)
            {
                arr[i][j]=sc.nextInt();
            }
        }
        int sum=0;
        for (int i = 0; i < arr.length; i++) {
        for (int j = 0; j < arr.length; j++) {
        if (i == j)
        sum = sum + arr[i][j];
        }}
       System.out.println(sum);
    }
}
```

program to analyze



constraint solver

mathematical constraints

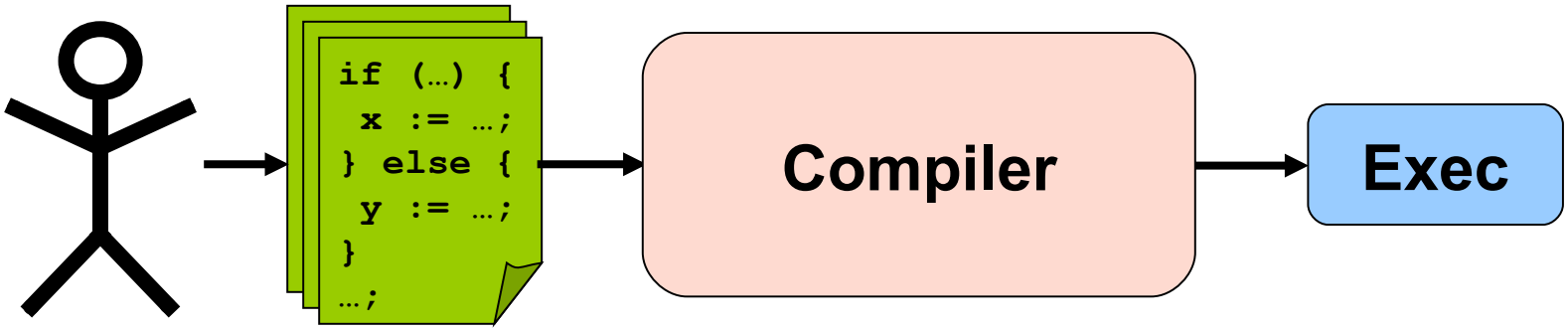
⟦p⟧ = &int
⟦q⟧ = &int
⟦alloc⟧ = &int
⟦x⟧ = ❄
⟦foo⟧ = ❄
⟦&n⟧ = &int
⟦main⟧ = ()->int

solution

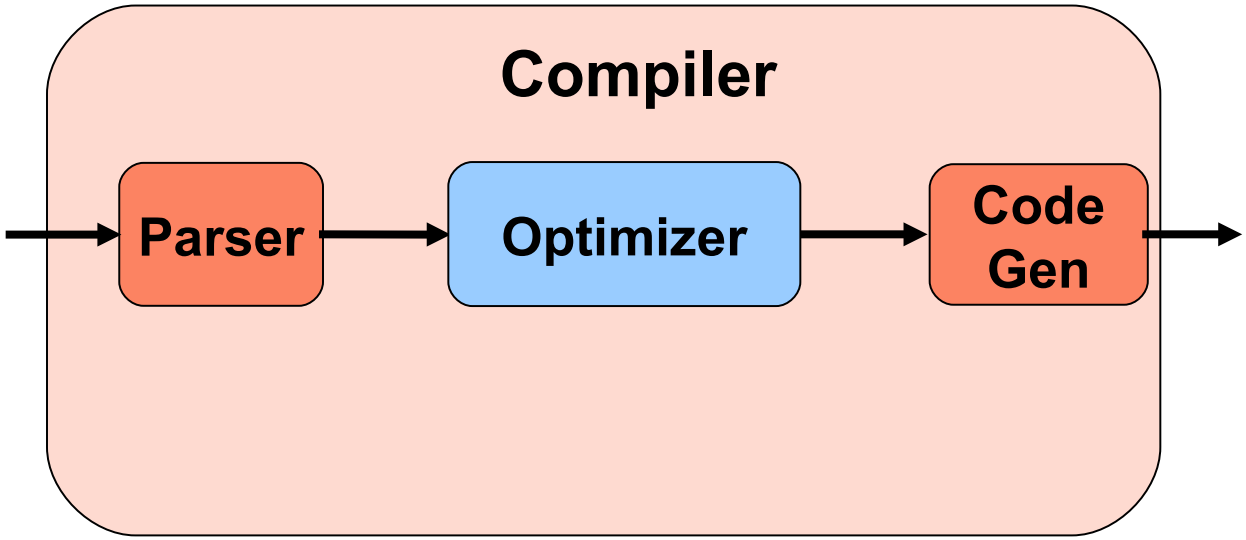
Challenging features in modern programming language

- Higher-order functions
- Mutable records or objects, arrays
- Integer or floating-point computations
- Dynamic dispatching
- Inheritance
- Exceptions
- Reflection
- …

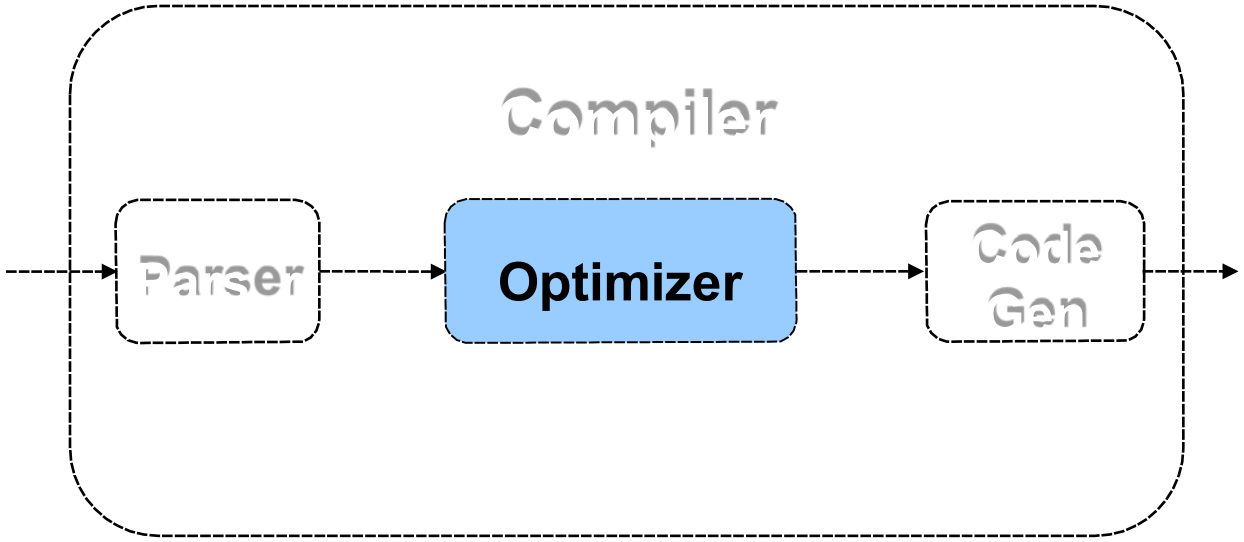
Let’s look at a compiler



Let’s look at a compiler

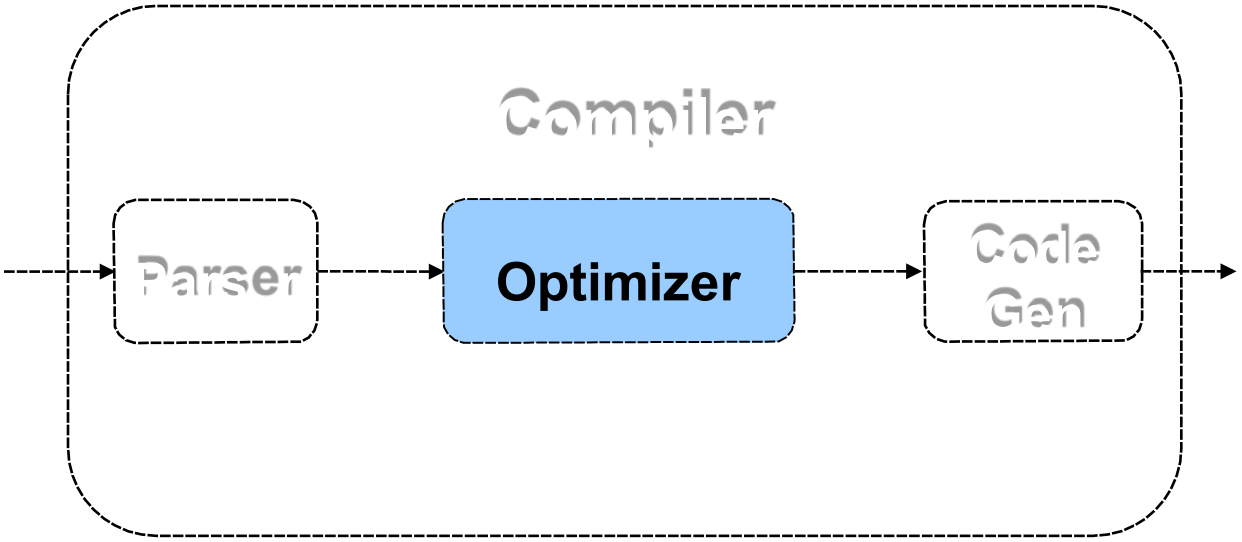


What does an optimizer do?



1. Analysis
2. Transformations

What does an optimizer do?



1. Compute information about a program
2. Use that information to perform program transformations

(with the goal of improving some metric, e.g. performance)

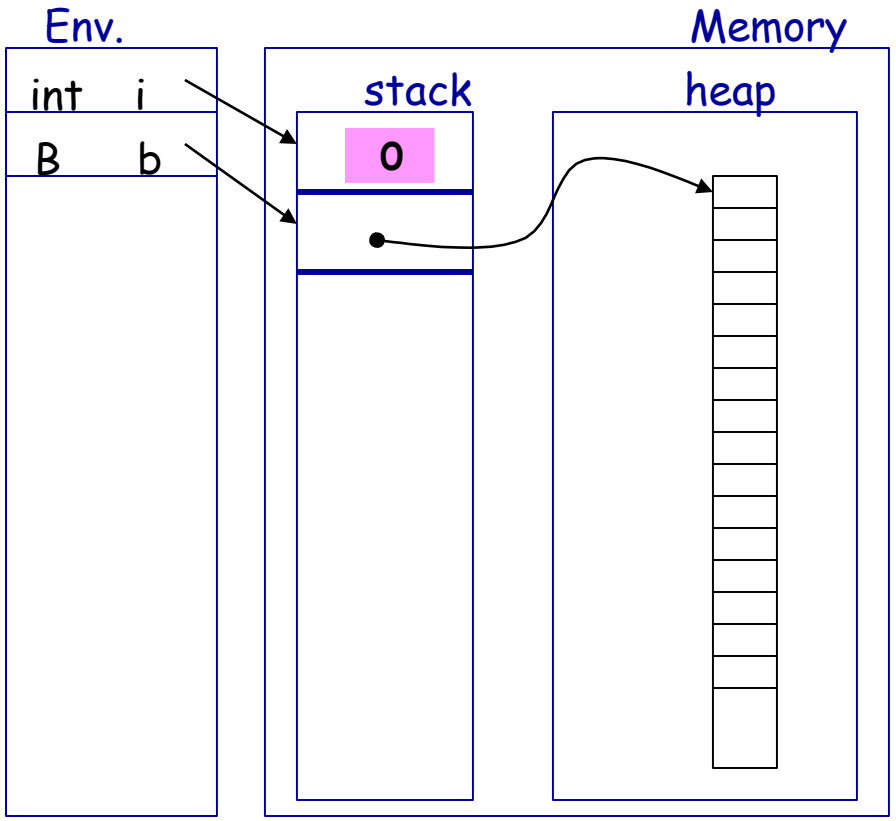
Example: Escape Analysis

- Consider these two C procedures:

```
typedef int A[10000];
typedef A* B;

void execute1(){
   int i;
   B b;
   b = (A*) malloc(sizeof(A));
   for (i=0; i<10000; i++)
        (*b)[i]=0;
   free(b);
}
```

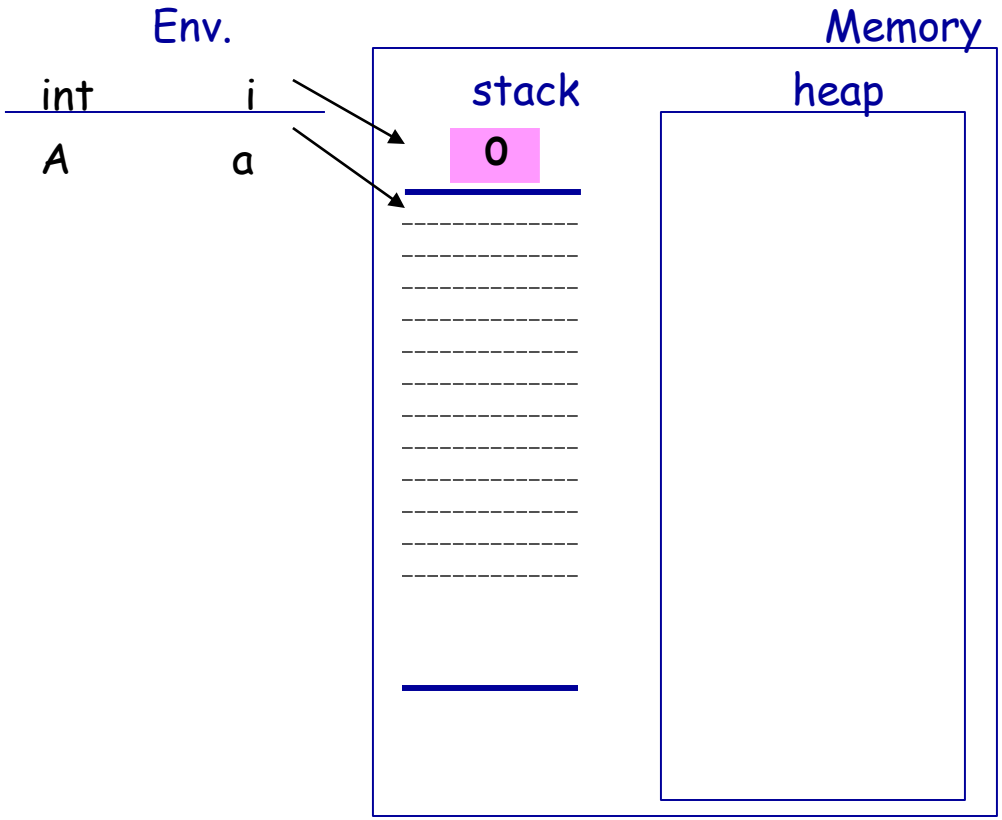
```
void execute2(){
   int i;
   A a;
   for (i=0; i<10000; i++)
        a[i]=0;
}
```

```
typedef int A[10000];
typedef A* B;
void execute1(){
   int i;
   B b;
   b = (A*) malloc(sizeof(A));
   for (i=0; i<10000; i++)
        (*b)[i]=0;
   free(b);
}
```



- The array b is allocated in the heap, and then de-allocated

```
typedef int A[10000];

void execute2(){
   int i;
   A a;
   for (i=0; i<10000; i++)
        a[i]=0;
}
```



- The array a is in the stack

Memory allocation has a cost…

```
for (i=0; i<1.000.000; i++)
    execute1();                 // about 90 sec

for (i=0; i<1.000.000; i++)
    execute2();                 // about 35 sec
```

- The second program is three times faster than the first one
- I f we can predict that a dynamic variable does not escape out of the procedure in which it is allocated, we can use a static variable instead, getting a more efficient program

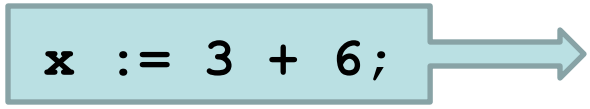
Overview of Optimizations

Simple example

```
foo(z) {
    x := 3 + 6;
    y := x - 5

    return z * y

}
```

Simple example

```
foo(z) {
   x := 3 + 6;
   y := x - 5;
   return z * y
}
```

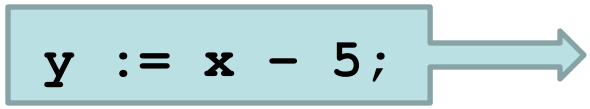


```
x:=9;
```

Applying Constant Folding

Simple example

```
foo(z) {

   x := 9;

   y := x - 5;

   return z * y

}
```

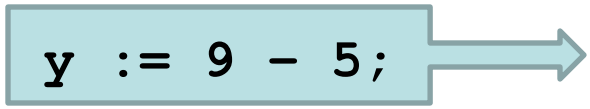


```
y:= 9 - 5;
```

By Constant Propagation

Simple example

```
foo(z) {
    x := 9;
    y := 9 - 5;
    return z * y
}
```

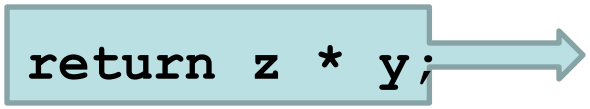


```
y:=4;
```

Applying Constant Folding

Simple example

```
foo(z) {
    x := 9;
    y := 4;
    return z * y;
}
```

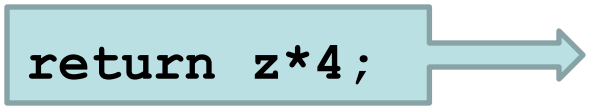


```
return z*4;
```

By Constant Propagation

Simple example

```
foo(z) {
    x := 9;
    y := 4;
    return z*4;
}
```



```
return z << 2;
```

By Strenght Reduction

Simple example

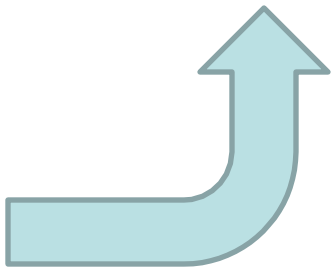
```
foo(z) {
   x := 9;
   y := 4;
   return z << 2;
}
```

By Dead Assignment Elimination

Simple example

```
foo(z) {

    x := 3 + 6;

    y := x - 5

    return z * y

}
```

```
foo(z) {

    return z << 2;

}
```



Constant Folding
Constant Propagation
Strenght Reduction
Dead Assignment Elimination

The TIP language

- ***T***iny ***I***mperative ***P***rogramming language

- Example language used in this course:
  - minimal C-style syntax
  - cut down as much as possible
  - enough features to make static analysis challenging and fun

- Scala implementation available

Expressions

$$
\begin{aligned}
E \rightarrow\ & I \\
\mid\ & X \\
\mid\ & E{+}E \mid E{-}E \mid E{*}E \mid E/E \mid E{>}E \mid E{==}E \\
\mid\ & (\,E\,) \\
\mid\ & \texttt{input}
\end{aligned}
$$

- *I* represents an integer constant
- *X* represents an identifier (x, y, z,…)
- `input` expression reads an integer from the input stream
- comparison operators yield 0 (false) or 1 (true)

Statements

$$
\begin{aligned}
S \rightarrow\ & X = E; \\
\mid\ & \texttt{output}\ E; \\
\mid\ & S\ S \\
\mid\ & \\
\mid\ & \texttt{if}\ (E)\ \{S\}\ [\texttt{else}\ \{S\}]^{?} \\
\mid\ & \texttt{while}\ (E)\ \{S\}
\end{aligned}
$$

- In conditions, 0 is false, all other values are true
- The `output` statement writes an integer value to the output stream

Functions

- Functions take any number of arguments and return a single value:

$$F \rightarrow X\ (\ X,\ \ldots,\ X\ )\ \{$$
$$\quad [\texttt{var}\ X,\ \ldots,\ X;]^{?}$$
$$\quad S$$
$$\quad \texttt{return}\ E;$$
$$\}$$

- The optional `var` block declares a collection of uninitialized variables
- Function calls are an extra kind of expressions:

$$E \rightarrow X\ (\ E,\ \ldots,\ E\ )$$

Records

$$E \rightarrow \{\, X : E,\ \ldots,\ X : E \,\}$$
$$\mid\ E.X$$

Pointers

$$
\begin{aligned}
E \rightarrow\ & \texttt{alloc}\ E \\
\mid\ & \&X \\
\mid\ & *E \\
\mid\ & \texttt{null}
\end{aligned}
$$

$$S \rightarrow\ *X = E;$$

(No pointer arithmetic)

Functions as values

- Functions are first-class values
- The name of a function is like a variable that refers to that function

- Generalized function calls:

$$E \rightarrow E(\,E\,,\,\ldots\,,\,E\,)$$

- Function values suffice to illustrate the main challenges with methods (in object-oriented languages) and higher-order functions (in functional languages)

Programs

- A program is a collection of functions
- The function named `main` initiates execution
  - its arguments are taken from the input stream
  - its result is placed on the output stream
- We assume that all declared identifiers are unique

$$P \rightarrow F \ldots F$$

An iterative factorial function

```
ite(n) {
  var f;
  f = 1;
  while (n>0) {
    f = f*n;
    n = n-1;
  }
  return f;
}
```

A recursive factorial function

```
rec(n) {
   var f;
   if (n==0) {
      f=1;
   } else {
      f=n*rec(n-1);
   }
   return f;
}
```

An unnecessarily complicated function

```
foo(p,x) {
  var f,q;
  if (*p==0) {
    f=1;
  } else {
    q = alloc 0;
    *q = (*p)-1;
    f=(*p)*(x(q,x));
  }
  return f;
}
```

```
main() {
  var n;
  n = input;
  return foo(&n,foo);
}
```

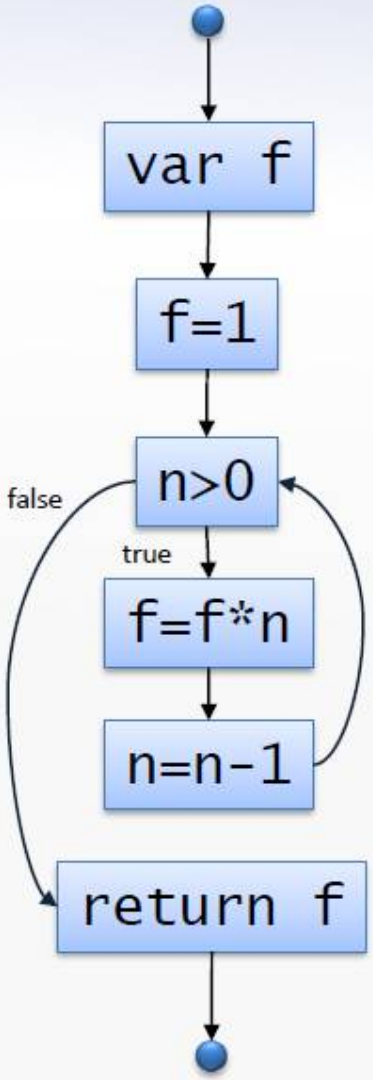
Beyond TIP

Other common language features in mainstream languages:

- global variables
- objects
- nested functions
- …

Control flow graphs

```
ite(n) {
  var f;
  f = 1;
  while (n>0) {
    f = f*n;
    n = n-1;
  }
  return f;
}
```



Control flow graphs

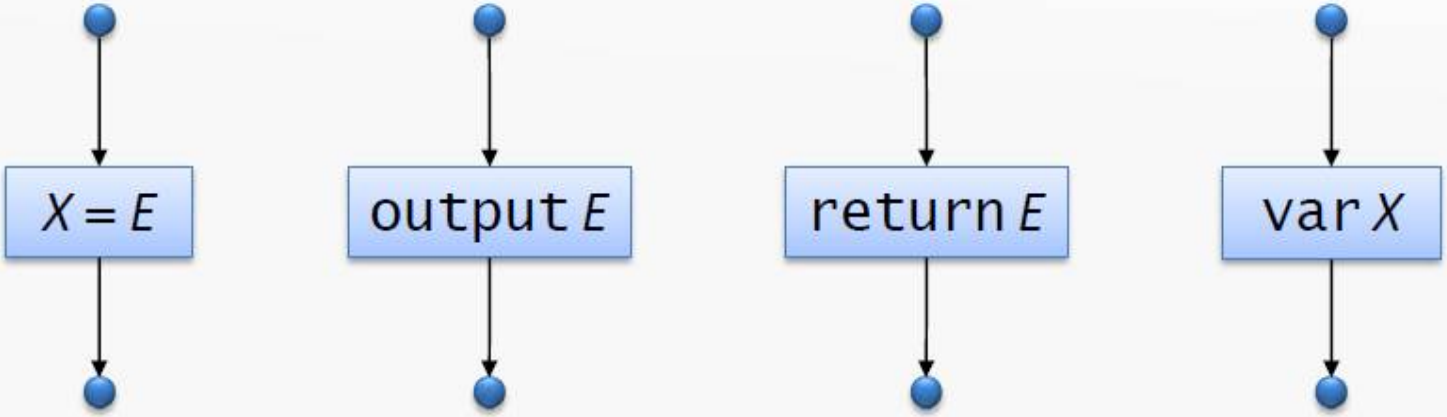
- A *control flow graph* (CFG) is a directed graph:
  - *nodes* correspond to program points (either immediately before or after statements)
  - *edges* represent possible flow of control
- A CFG always has
  - a single point of *entry*
  - a single point of *exit*

  (think of them as no-op statements)
- Let v be a node in a CFG
  - *pred*(v) is the set of predecessor nodes
  - *succ*(v) is the set of successor nodes

CFG construction (1/3)

- For the simple `while` fragment of TIP, CFGs are constructed inductively

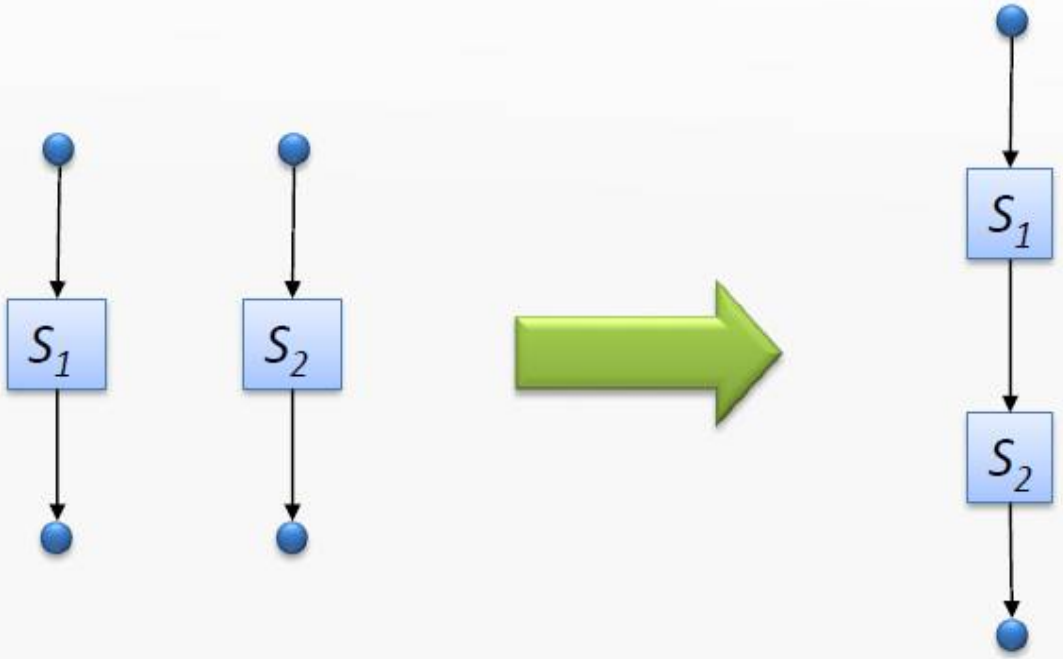
- CFGs for simple statements etc.:



CFG construction (2/3)

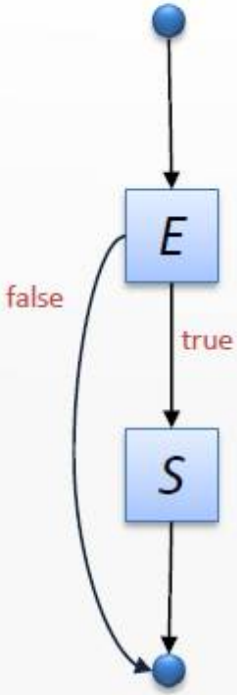
For a statement sequence S_1 S_2:

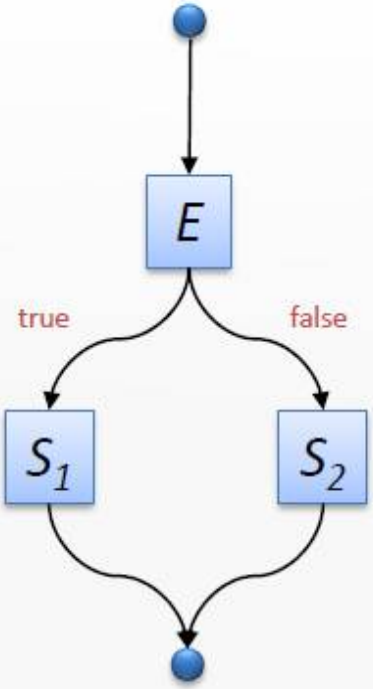
- eliminate the exit node of S_1 and the entry node of S_2
- glue the statements together

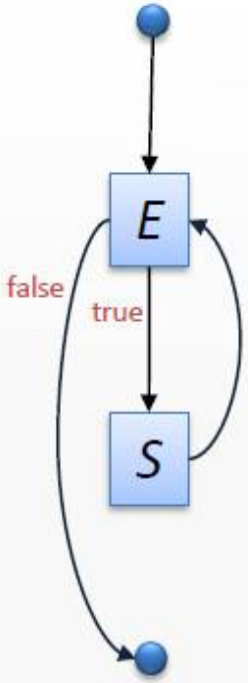


CFG construction (3/3)

Similarly for the other control structures:







Normalization

- Sometimes convenient to ensure that each CFG node performs only one operation
- *Normalization*: flatten nested expressions, using fresh variables

```
x = f(y+3)*5;
```

```
t1 = y+3;
t2 = f(t1);
x = t2*5;
```

Homework

Please read Chapter 1 and Chapter 2 from the book

https://cs.au.dk/~amoeller/spa/spa.pdf