

Comps review session

COMPILERS

Justin Talbot
Isil Ozgener
Thomas Dillig

Front End

Lexical
analysis

Regular
expressions

Tokenize

Parsing

Context-free

Build parse tree
and AST

Semantic
analysis

Other

Type checking
Scoping

“if a == b then”



<if> <id> <eq> <id> <then>

abcd

-ab

-abc*

abcd

(1) ab

(2) abc*

abcd

(1) ab

(2) abc*

Rules:

Maximal match

Use ordering to break ties

abcd

(1) ab

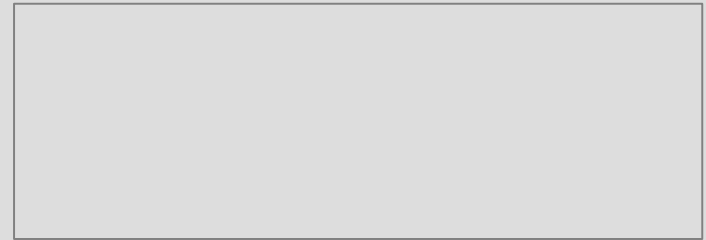
(2) abc*



Top down parsing

$S \rightarrow cAd$

$A \rightarrow ab \mid a$



cad

Top down parsing

$S \rightarrow cAd$

$A \rightarrow ab \mid a$

Start

S

cad

Top down parsing

$S \rightarrow cAd$

$A \rightarrow ab \mid a$

Start



Guess

S

cad

Top down parsing

$S \rightarrow cAd$

$A \rightarrow ab \mid a$

Start



Guess

cAd

cad

Top down parsing

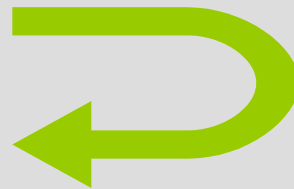
$S \rightarrow cAd$

$A \rightarrow ab \mid a$

Start



Guess



cabd

cad

Top down parsing

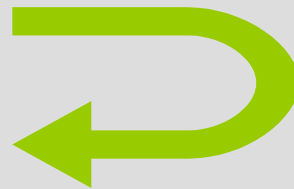
$S \rightarrow cAd$

$A \rightarrow ab \mid a$

Start



Guess



Backtrack



cabd

cad

Top down parsing

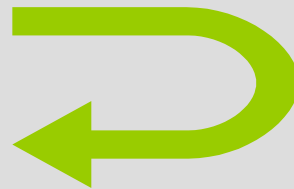
$S \rightarrow cAd$

$A \rightarrow ab \mid a$

Start



Guess



cAd

cad

Top down parsing

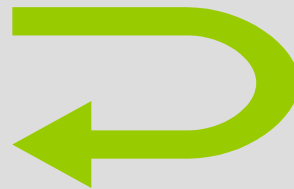
$S \rightarrow cAd$

$A \rightarrow ab \mid a$

Start



Guess



cad

cad

Top down parsing

$S \rightarrow cAd$

$A \rightarrow ab \mid a$

Start



Guess



Accept

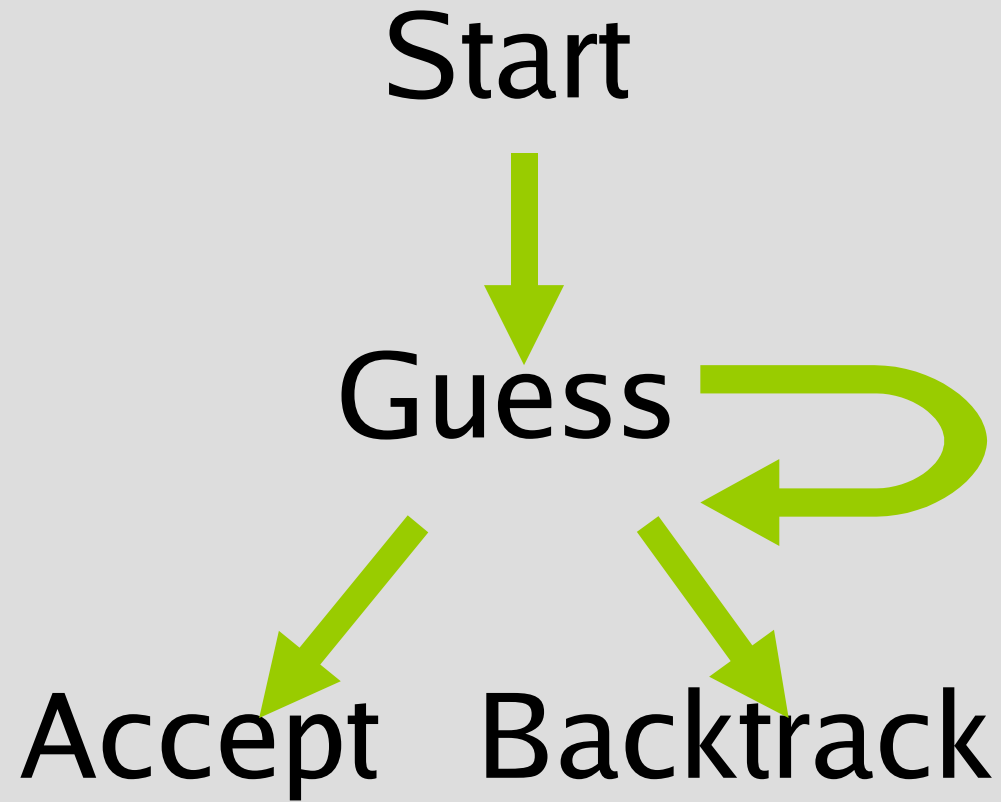
cad

cad

Top down parsing

$S \rightarrow cAd$

$A \rightarrow ab \mid a$





Left recursion

$$A \rightarrow Aa \mid \beta$$

Left recursion

$$A \rightarrow Aa \mid \beta$$



$$A \rightarrow \beta A'$$

$$A' \rightarrow aA' \mid \varepsilon$$

Left factoring

$$B \rightarrow aC \mid a\beta$$

Left factoring

$$B \rightarrow a\beta \mid aC$$



$$B \rightarrow aB'$$

$$B' \rightarrow \beta \mid C$$

First sets
Follow sets



Shift-reduce parsing

Structure of parser

- Input
- Stack
- State machine
- Action and goto tables

Shift-reduce conflict

Reduce-reduce conflict

To resolve:

- Change grammar
- Modify parser to either shift or reduce

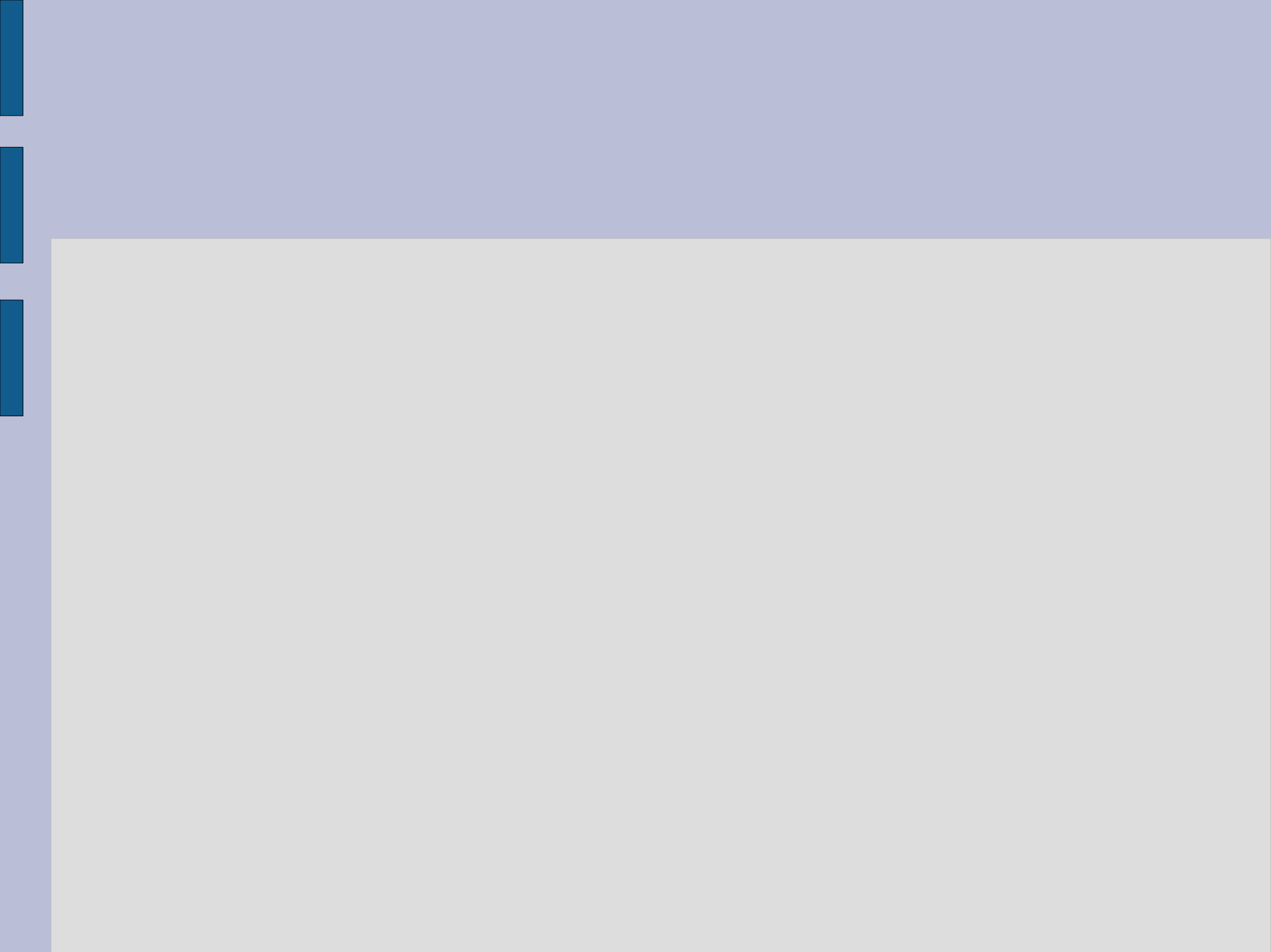
Error reporting

Panic mode

Syntax error recovery

Correction

Canonical collections Example



Abstract Syntax Tree

“Semantic actions”

run by parser on reduce

Parse tree has too much information

Type Checking

- What do we mean by a type?
Type is a a set of values and a set of operations defined on those values.
- The goal of type checking is to ensure that operations are used with the correct set of values.
- Only certain operations make sense on certain types. For example, it doesn't make sense to add a function pointer and an integer.
- By type checking a program, we can ensure that operations have their intended interpretation.

Static vs Dynamic Typing

- Type checking can be done both statically (e.g. C, Java etc) or dynamically (e.g Scheme).
- Static typing: Type checking done during compilation.
- Dynamic typing: Type checking is done at run-time.
- Trade-offs:
 - Dynamic:
 - Con: Overhead during runtime
 - Con: Run-time exceptions
 - Static:
 - Con: More restrictive than dynamic typing.
 - Usually mechanism to escape these restrictions (such as casts)

Type Checking vs Type Inference

- Hindley-Miller type inference: Given a term of untyped lambda calculus, type inference finds **all** terms of typed lambda calculus which yield the given term when type information on bound variables is deleted. (used in ML, Haskell, OCaml)
- Type checking is different from type inference: Type checking merely **verifies** fully typed programs, it does not infer possible types for untyped expressions.

Type Environment

- **Type environment:** A mapping between object identifiers to their corresponding types.
- A type environment gives types for free variables. A variable is free in an expression if it is not defined within that expression.
- Notation:
“ $O \vdash e:T$ ” means that in type environment O , it is provable that expression e has type T .

Type Checking Rules

- Type checking rules are written as logic rules, with a set of hypotheses and a conclusion.

- Example:

$O \vdash e1:\text{int}$

$O \vdash e2:\text{int}$

$O \vdash e1+e2:\text{int}$

- Another example:

$O[T0/x] \vdash e1:T1$

$O \vdash \text{let } x:T0 \text{ in } e1$

Soundness

- A type system is sound if:
 - Whenever $O \vdash e:T$, then under the assumptions given by O , e indeed evaluates to T .

CODE GENERATION

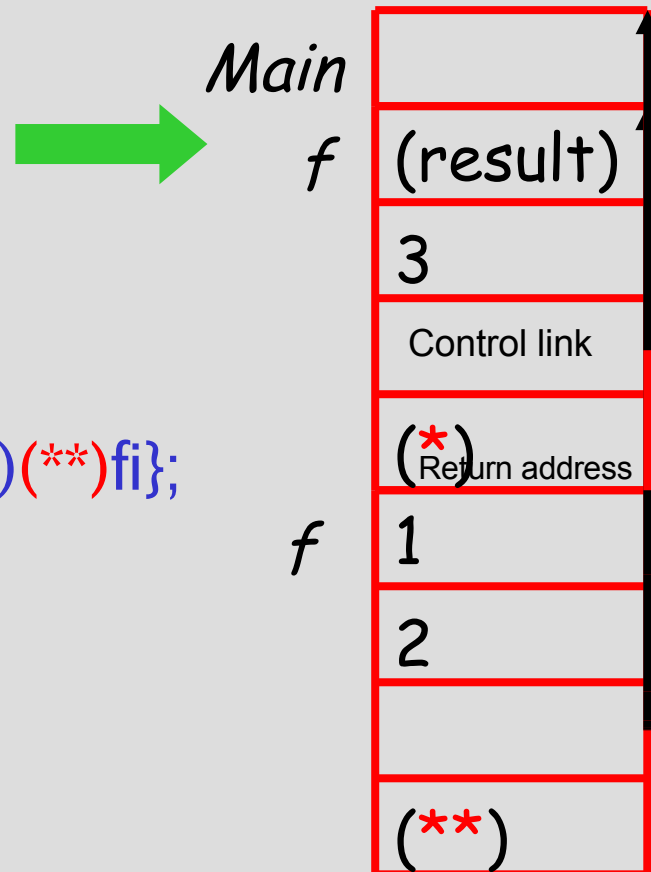
Activation Records

- The information needed to manage one procedure activation is called an *activation record (AR)* or *frame*
- Consider the program:

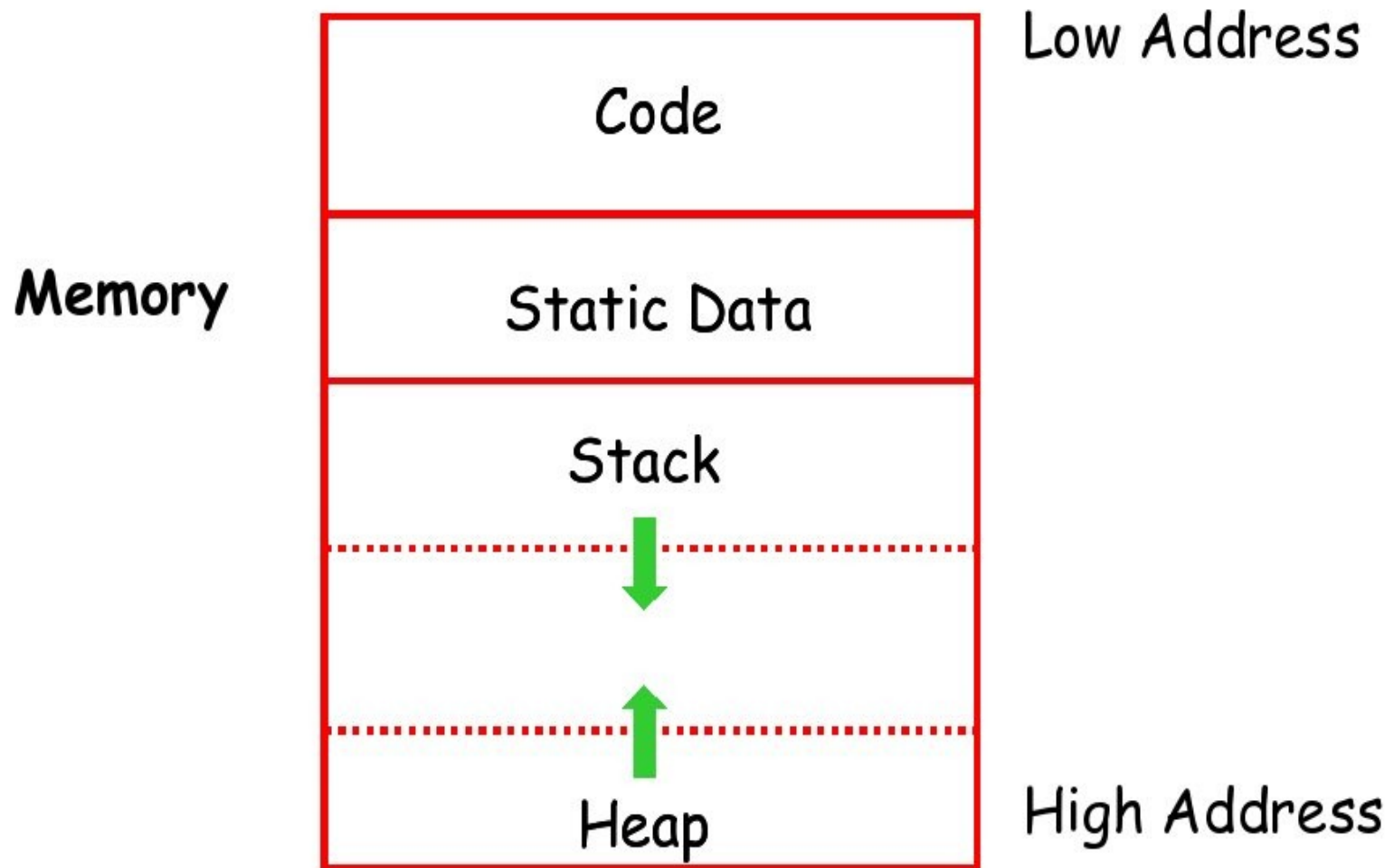
```
g() : Int { 1 };
```

```
f(x:Int):Int {if x=0 then g() else f(x - 1)(**)fi};
```

```
main(): Int {{f(3); (*}  
}}
```



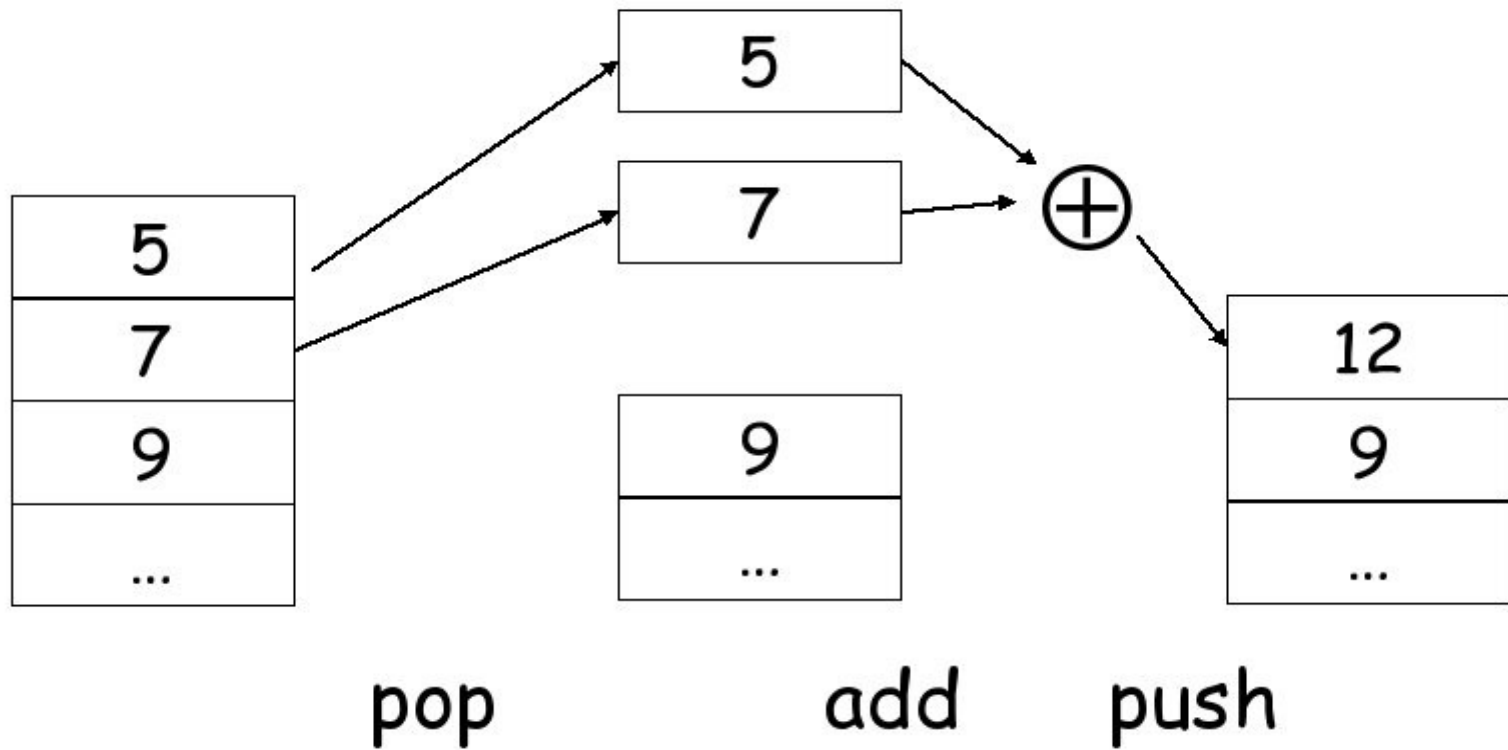
Memory Layout



Stack Machines

- A simple evaluation model for instructions
- Each instruction:
 - Takes its operands from the stack
 - Removes operands from the stack
 - Computes the required operation
 - Pushes result on the stack
- Advantages
 - Location of the operands is implicit: on top of stack
 - No need to worry about register allocation
 - Every instruction preserves the contents of the stack before this instruction (no need for control link)
 - Simplifies code generation

Add instruction with Stack Machine



Code Generation

- It's easy to encode one stack machine operation as a sequence of:
 - Loading operands from the stack
 - Performing the instruction
 - Storing the result on the stack.
- Example: (RED is compile time, BLUE is run-time)

cgen($e_1 + e_2$) =

cgen(e_1)

sw \$a0 0(\$sp) /*store content of a0 to the top of the stack*/

addiu \$sp \$sp -4 /* decrement stack pointer */

cgen(e_2)

lw \$t1 4(\$sp) /*Load second elem of stack to register \$t1*/

add \$a0 \$t1 \$a0 /*Add top of stack and t1, result in a0*/

addiu \$sp \$sp 4 /* Restore the original stack pointer */

OPTIMIZATIONS

When do we optimize?

- How about at the AST level?
 - Pro: Completely machine independent
 - Con: Too high-level to be useful for optimizing
- How about at the assembly level?
 - Pro: Low-level; so we can optimize
 - Con: Machine dependent – must reimplement to target different machines
- How about something in between, such as machine-independent “high-level assembly”?
 - Better solution because it is both machine independent and exposes optimization opportunities.

Intermediate Language (IL)

- “High-level” assembly:
 - Unlimited number of registers, so don't need to worry about register allocation
 - Higher level opcodes. E.g: Push translates into many opcodes, but we can allow it in IL.
- A common form of IL is “Three Address Code.”
 - Every instruction is of the form $z = x \text{ op } y$
 - x, y, z are registers (or constants).
 - Example: Translate $d = a*b - c$ to TAC:
 - $e = a*b$
 - $d = e - c$

Some Basic Concepts

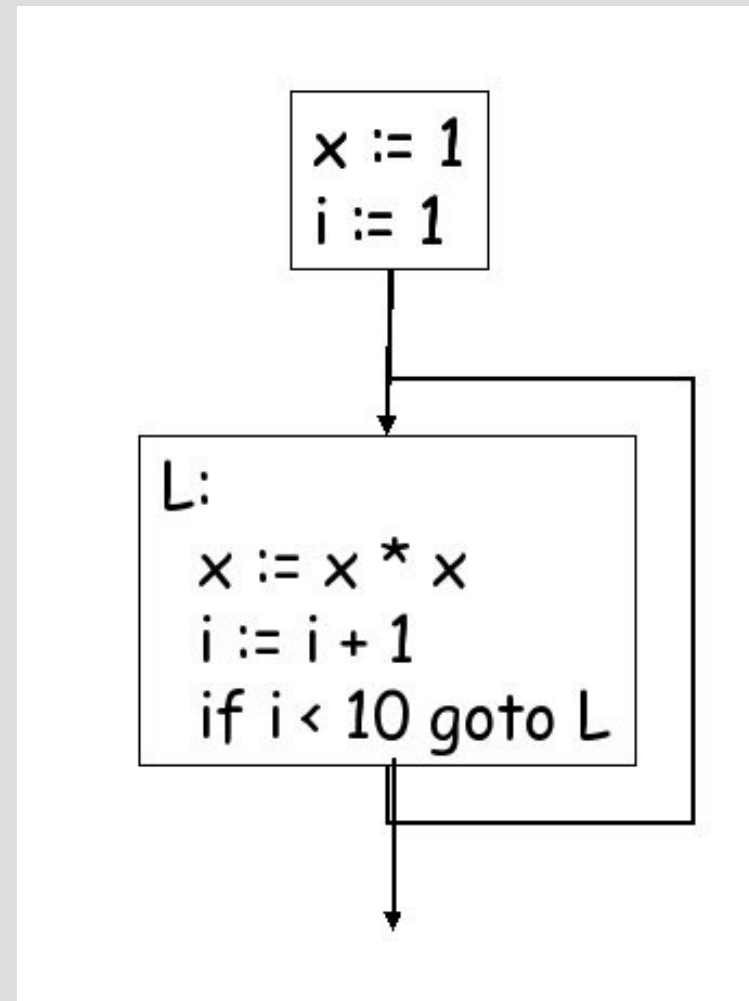
- **Basic blocks:**
 - Maximal sequence of instructions with no labels or jumps (instructions usually in IL for optimizations)
 - Has exactly one entry and one exit point
 - Only last instruction can involve control flow

Basic Concepts Continued

- **Control Flow Graph (CFG):**
 - A directed graph representing a function body with basic block as nodes.
 - A directed edge from block A to B indicates that execution can pass from last instruction of A to first instruction of B (but not in the middle)
 - Fundamental representation for optimizations and dataflow

Control Flow Graph Example

- CFG for program that computes x^{10} . with $x=1$ in this example.
- Note that the edge from L to L indicates that execution reaches L if $i < 10$.



Flavors of Optimizations

- There are three levels of optimizations:
 - **Local:** Optimizes only within basic blocks
 - **Global:** Optimizes within a function (CFG)
 - **Interprocedural:** Optimizes across function boundaries
- As we go from local to interprocedural, the cost of optimization increases and gets more and more complicated.
- Interprocedural optimizations are almost never done.

Local Optimization

- Many simple local optimizations are possible, we'll only discuss algebraic simplification, constant folding, common subexpression elimination, and copy propagation.
- **Algebraic simplification:**
 - Translate into cheaper instructions. E.g: Use $x*x$ instead of x^2 .
- **Constant folding:**
 - If the value of a constant can be computed at compile time, do so. E.g. Rewrite $x = 2 + 2$ as $x=4$.

More Local Optimizations

- **Common subexpression elimination:**
 - If two expressions share a common subexpression, compute it only once and use it in both expressions.
- **Copy propagation:**
 - If there is an assignment $x = z$, replace x by z for all uses of x .
- Important note: The latter two optimizations require that variable names to be used exactly once. This can be achieved by rewriting a function in single assignment form (SAS).

Global Optimizations

- Common framework for global optimizations is data-flow analysis.
- Data-flow is a general technique that allows us to reason about properties at a given point in the program.

Dataflow Framework

- Data-flow analysis involves writing a set of **data-flow equations** for each node of the CFG and computing the output from the input (or vice versa) until the system stabilizes (reaches a fixpoint).
- If the output is computed from the input, we call this a **forward analysis**; if input is computed from output, we call this a **backward analysis**.

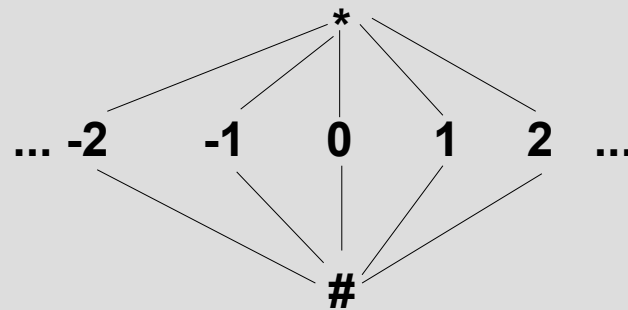
Global Constant Propagation

- A typical example of forward data flow analysis global copy propagation.
- Given a variable X at a given point, we want to know if X is a constant or not, and if it is a constant, we want to compute its value at compile time.
- But we must be always be conservative: If X is not a constant and our analysis concludes it is, the program will be incorrect. So, it's OK to say that we “don't know” and assume that X is not a constant.

Constant Propagation: Dataflow Values

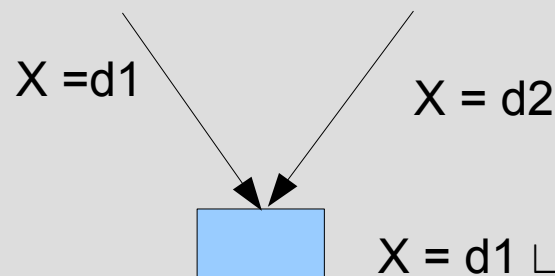
- Every variable X at every point in the program has a dataflow value associated with it:

- $\#$: not computed
- c : constant
- $*$: not a constant



Order:
 $* \geq c \geq \#$

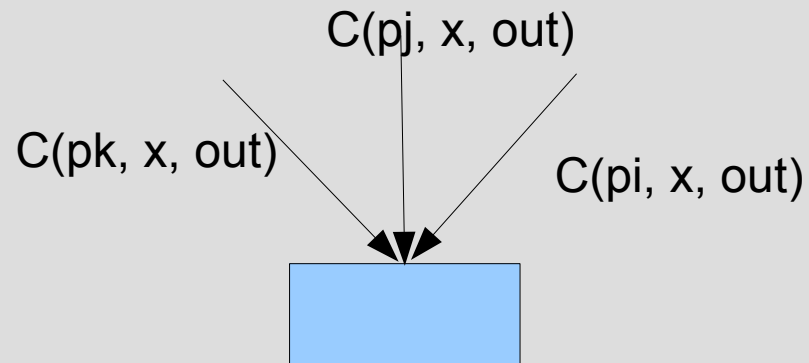
- Dataflow values must have a partial order defined on them to perform join.



The Dataflow Equations

- We now go through a complete example, writing the dataflow equations associated with each statement.
- These equations are called “transfer functions”.

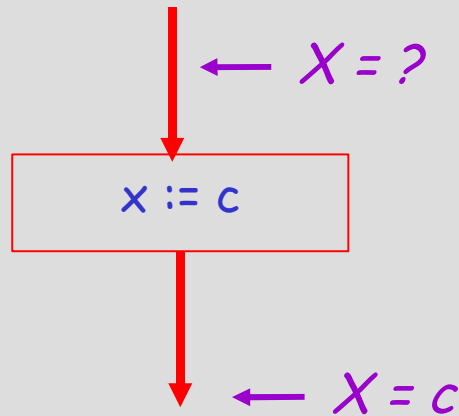
The Dataflow Equations



- $$C(p, x, \text{in}) = \bigsqcup_{p_i \in \text{pred}(p)} C(p_i, x, \text{out})$$

-

Dataflow Equations Continued



$C(x := c, x, \text{out}) = c$ if c is a constant

$C(s, x, \text{out}) = C(s, x, \text{in})$ if x is not an assignment to x

$C(x := f(\dots), x, \text{out}) = *$

The Algorithm

- Algorithm is simple:
- For every variable x , set its initial value to $*$ for the entry point in the function entry point and to $\#$ for other points, and apply the above equations until a fixed point is reached.