# Compilation

0368-3133

Course summary: Putting it all together

Noam Rinetzky

# The Exam

מרצה: **נעם רינצקי** מתרגל: **אורן איש שלום** חומר: **פתוח** משך: **שלוש שעות**

**<u>מבחן בקומפילציה – 2017/18 – מועד א</u>**

- המבחן מורכב מחמש שאלות. **יש לענות על כולן**.
- מומלץ לקרוא את השאלה עד סופה לפני שמתחילים לענות.
- משקל השאלה ומספרה אינה מעיד על הקושי בפתירתה.
- **יש לציין בראש העמוד את השאלה עליה עונים.**
- **אין לענות על שאלות שונות באותו העמוד.**
- תשובה "איני יודע/ת" תזכה ב 20% מהניקוד על הסעיף הרלוונטי .

# Course Goals

- What is a compiler
- How does it work
- (Reusable) techniques & tools

# What is a Compiler?

"A compiler is a computer program that transforms source code written in a programming language (source language) into another language (target language).

The most common reason for wanting to transform source code is to create an executable program."

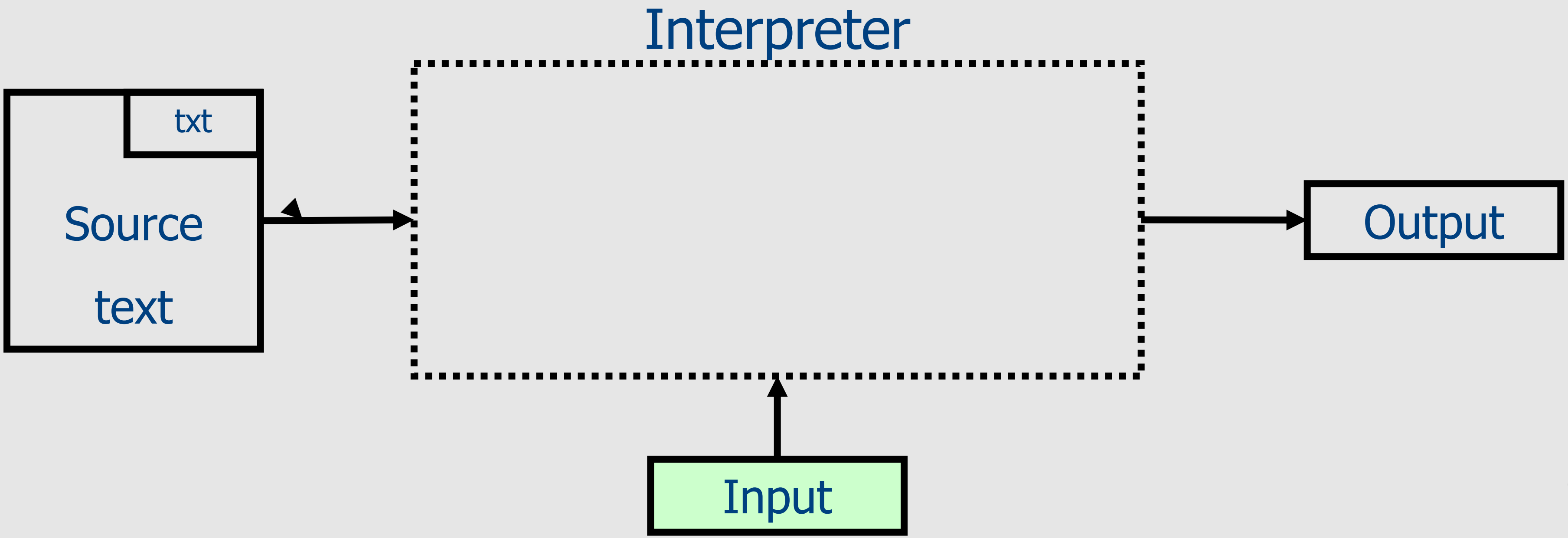
--*Wikipedia*

# Compiler

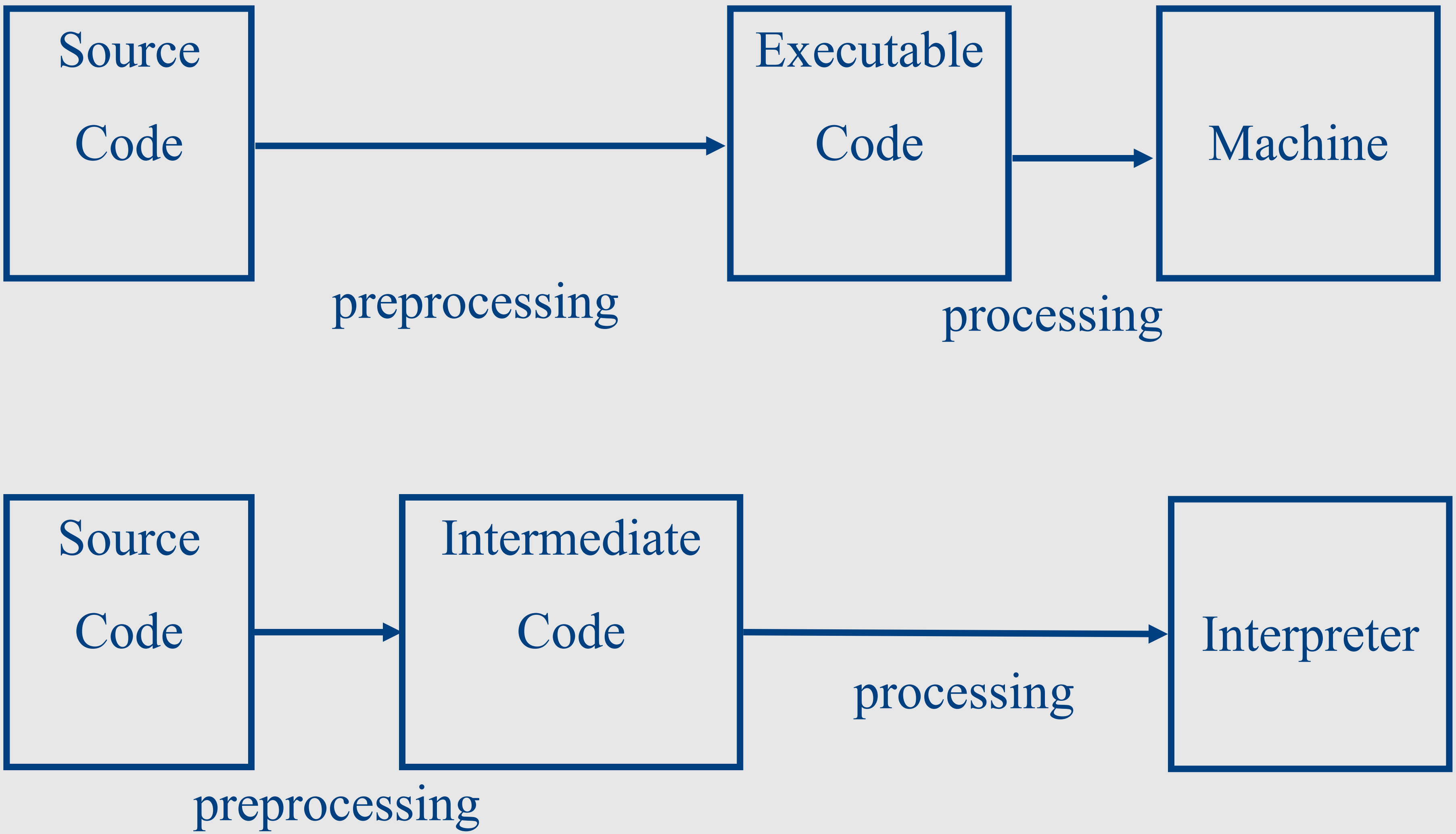
- A program which transforms programs
- Input a program (P)
- Output an object program (O)
  - For any x, “O(x)” “=“ “P(x)”

# Interpreter

- A program which executes a program
- Input a program (P) + its input (x)
- Output the computed output (P(x))



# Compiler vs. Interpreter



# Interpreter vs. Compiler

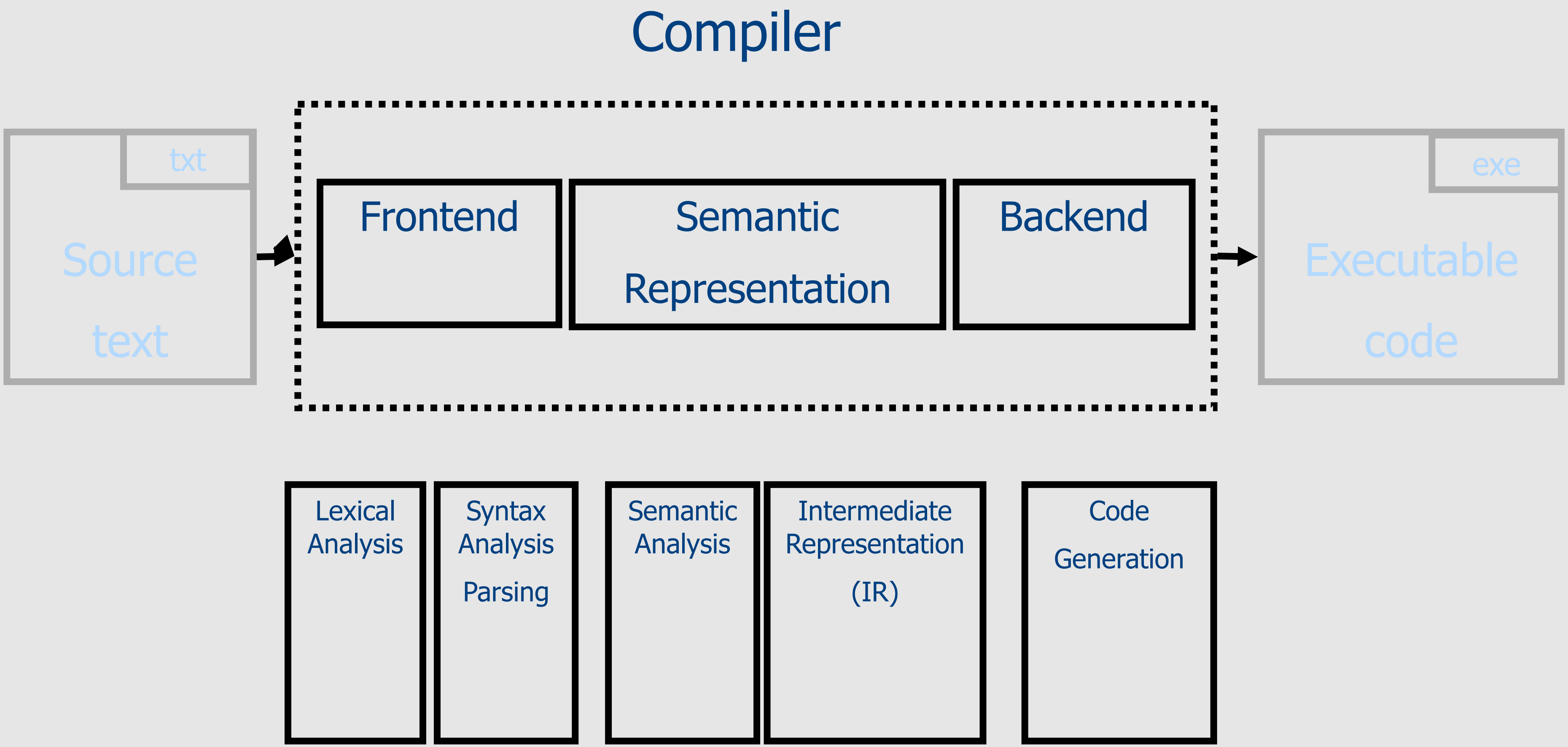
**Interpreter**

- Conceptually simpler
  - "define" the prog. lang.
- Can provide more specific error report
- Easier to port
- Faster response time
- [More secure]

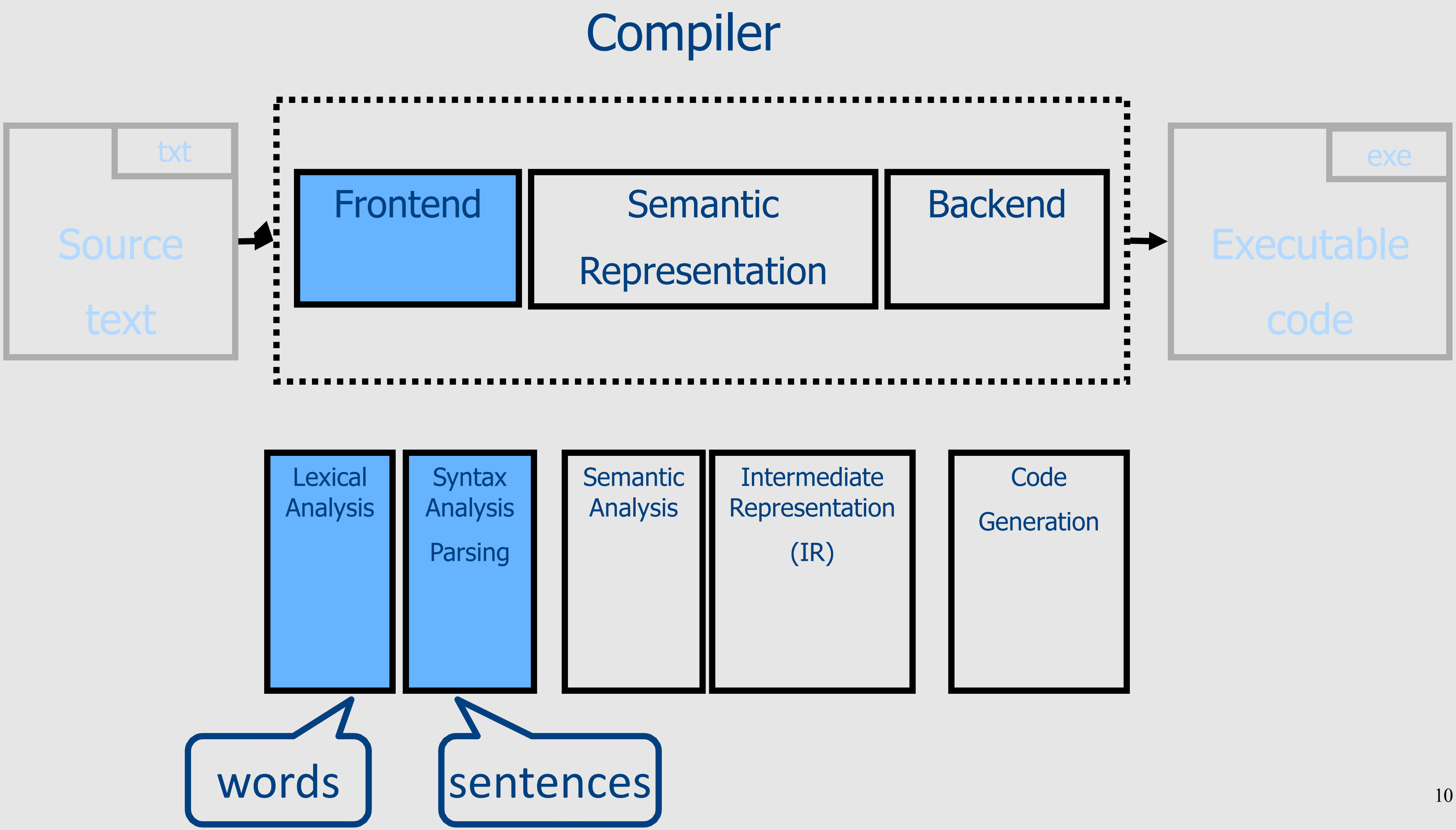
**Compiler**

- How do we know the translation is correct?
- Can report errors before input is given
- More efficient code
  - Compilation can be expensive
  - move computations to compile-time
- *compile-time + execution-time < interpretation-time* is possible

# Conceptual Structure of a Compiler



# Conceptual Structure of a Compiler



# Lexical Analysis

# What does Lexical Analysis do?

- Partitions the input into stream of tokens
  - Numbers
  - Identifiers
  - Keywords
  - Punctuation

> - “word” in the source language
> - “meaningful” to the syntactical analysis

- Usually represented as (kind, value) pairs
  - (Num, 23)
  - (Op, ‘*’)

# Some basic terminology

- Lexeme (aka symbol) - a series of letters separated from the rest of the program according to a convention (space, semi-column, comma, etc.)

- Pattern - a rule specifying a set of strings. Example: “an identifier is a string that starts with a letter and continues with letters and digits”
  - (Usually) a regular expression

- Token - a pair of (pattern, attributes)

# Regular languages

- Formal languages
  - Σ = finite set of letters
  - Word = sequence of letter
  - Language = set of words
- Regular languages defined equivalently by
  - Regular expressions
  - Finite-state automata

# From regular expressions to NFA

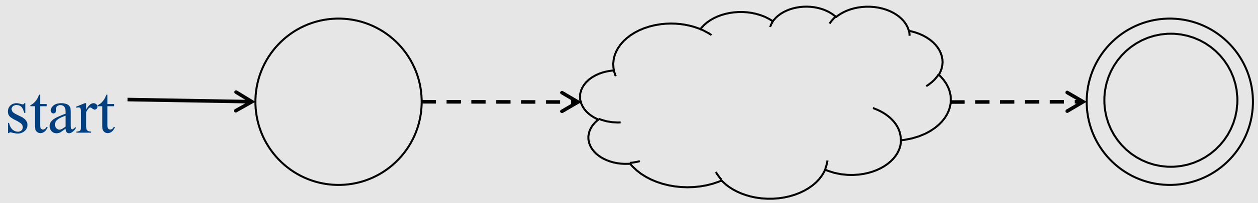
- Step 1: assign expression names and obtain pure regular expressions $R_1 \ldots R_m$


- Step 2: construct an NFA $M_i$ for each regular expression $R_i$
- Step 3: combine all $M_i$ into a single NFA

- *Ambiguity resolution: prefer longest accepting word*

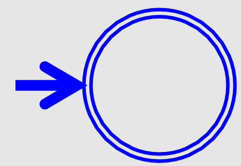
# From reg. exp. to automata

- Theorem: *there is an algorithm to build an NFA+Є automaton for any regular expression*
- Proof: *by induction on the structure of the regular expression*

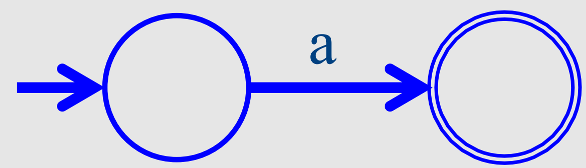


# Basic constructs

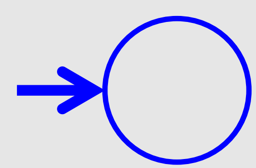
$R = \varepsilon$



$R = a$

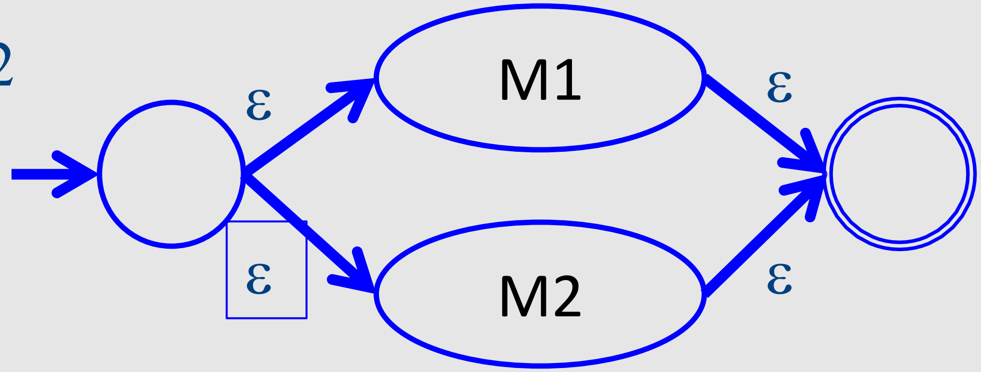



$R = \phi$

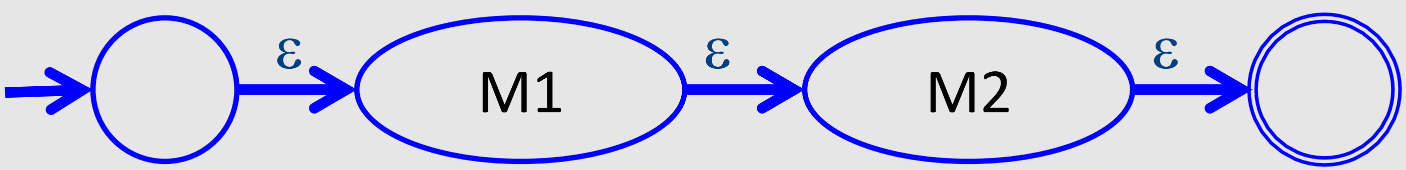


# Composition

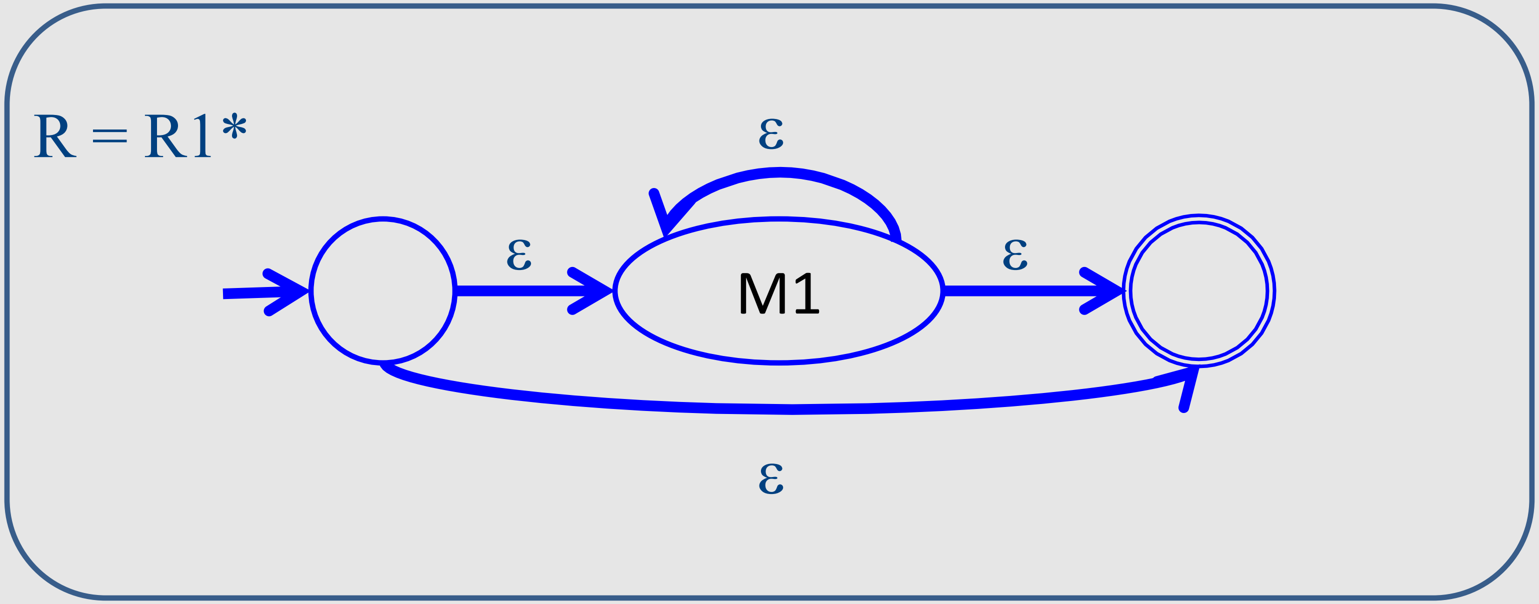
R = R1 | R2




R = R1R2



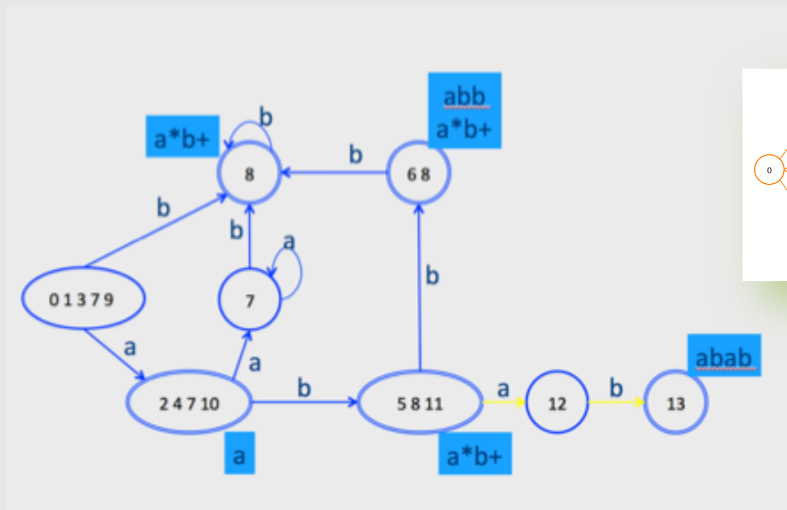
# Repetition

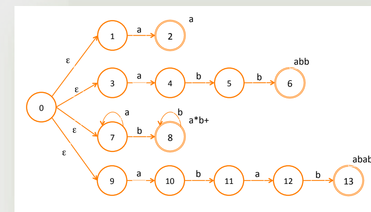


# Scanning with DFA

- Run until stuck
  - **Remember last accepting state**
- Go back to accepting state
- Return token

# Ambiguity resolution



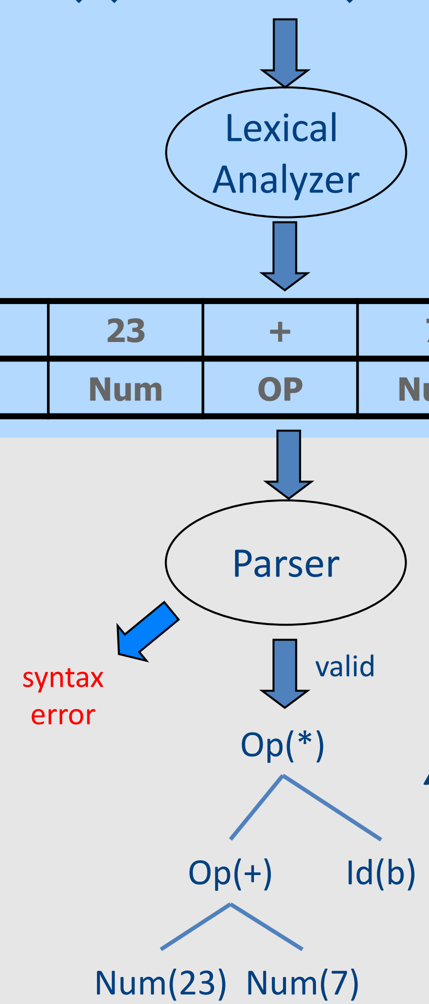




- Longest word
- Tie-breaker based on **order of rules** when words have same length

# Syntax Analysis

# Frontend: Scanning & Parsing

*program text* **`((23 + 7) * x)`**




*token stream*

| ( | ( | 23 | + | 7 | ) | * | x | ) |
|---|---|---|---|---|---|---|---|---|
| LP | LP | Num | OP | Num | RP | OP | Id | RP |

*Abstract Syntax Tree*

Grammar:

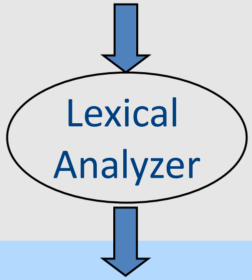
E → … | Id

**Id** → ‘a’ | … | ‘z’

# From scanning to parsing

*program text*

**((23 + 7) * x)**



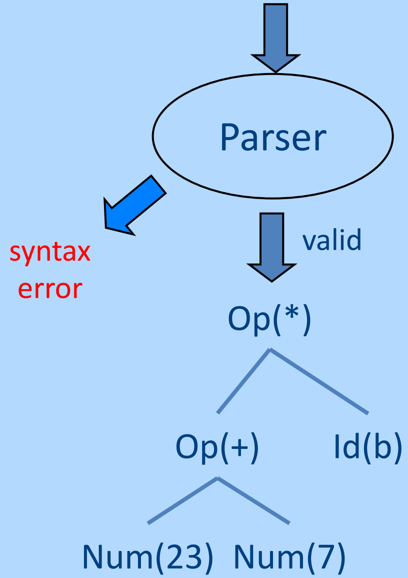

*token stream*

| ( | ( | 23 | + | 7 | ) | * | x | ) |
|---|---|---|---|---|---|---|---|---|
| LP | LP | Num | OP | Num | RP | OP | Id | RP |

Grammar:

E → ... | Id

**Id** → 'a' | ... | 'z'




*Abstract Syntax Tree*

# Context free grammars (CFG)

$G = (V,T,P,S)$

- **V** – non terminals (syntactic variables)
- **T** – terminals (tokens)
- **P** – derivation rules
  - Each rule of the form $V \rightarrow (T \cup V)^*$
- **S** – start symbol

# Pushdown Automata (PDA)

- Nondeterministic PDAs define all CFLs
- Deterministic PDAs model parsers.
  - Most programming languages have a deterministic PDA
  - Efficient implementation

# CFG terminology

- Derivation - a sequence of replacements of non-terminals using the derivation rules
- Language - the set of strings of terminals derivable from the start symbol
- Sentential form - the result of a partial derivation
  - May contain non-terminals

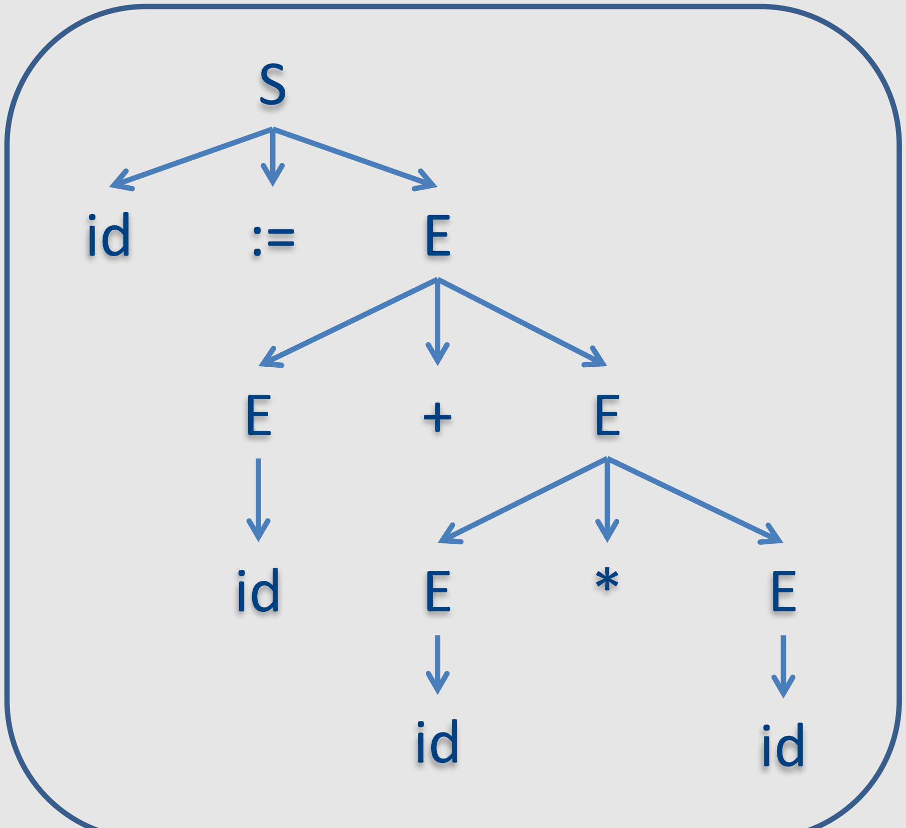
# Derivations

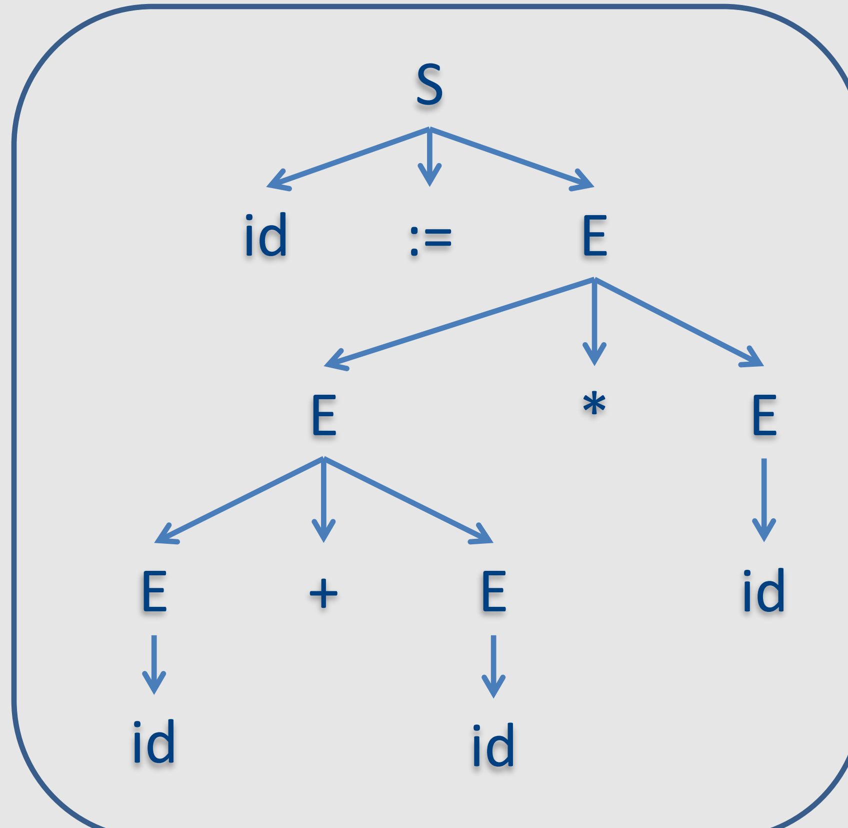
- Show that a sentence ω is in a grammar G
  - Start with the start symbol
  - Repeatedly replace one of the non-terminals by a right-hand side of a production
  - Stop when the sentence contains only terminals
- Given a sentence $\alpha N \beta$ and rule $N \rightarrow \mu$
  $\alpha N \beta \Rightarrow \alpha \mu \beta$
- ω is in L(G) if $S \Rightarrow^* \omega$

# Ambiguity

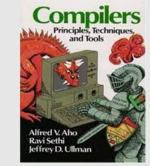
x := y+z*w

```
S → S ; S
S → id := E | …
E → id | E + E | E * E | …
```



# “dangling-else” example




p. 174

**Ambiguous grammar**

$S \rightarrow$ **if** E **then** S
S | **if** E **then** S **else** S
| other

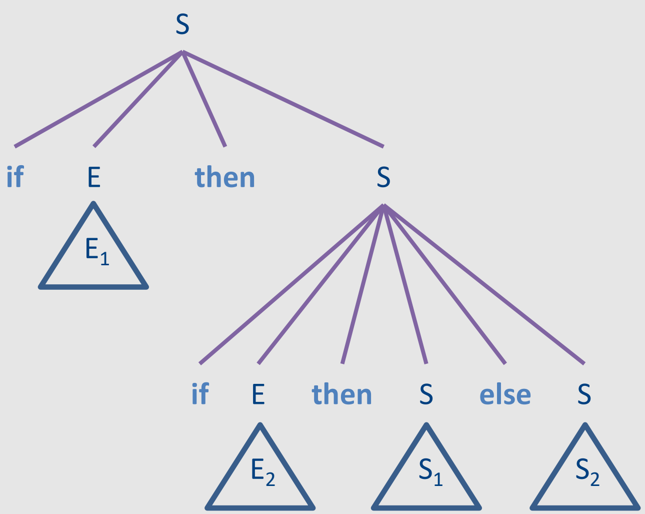
This is what we usually want: match **else** to closest unmatched **then**

**Unambiguous grammar**

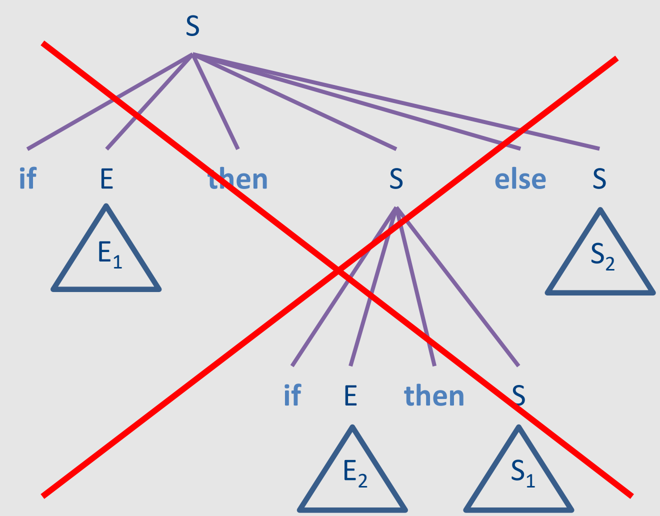
?

**if** $E_1$ **then if** $E_2$ **then** $S_1$ **else** $S_2$

**if** $E_1$ **then (if** $E_2$ **then** $S_1$ **else** $S_2$**)**




**if** $E_1$ **then (if** $E_2$ **then** $S_1$**) else** $S_2$

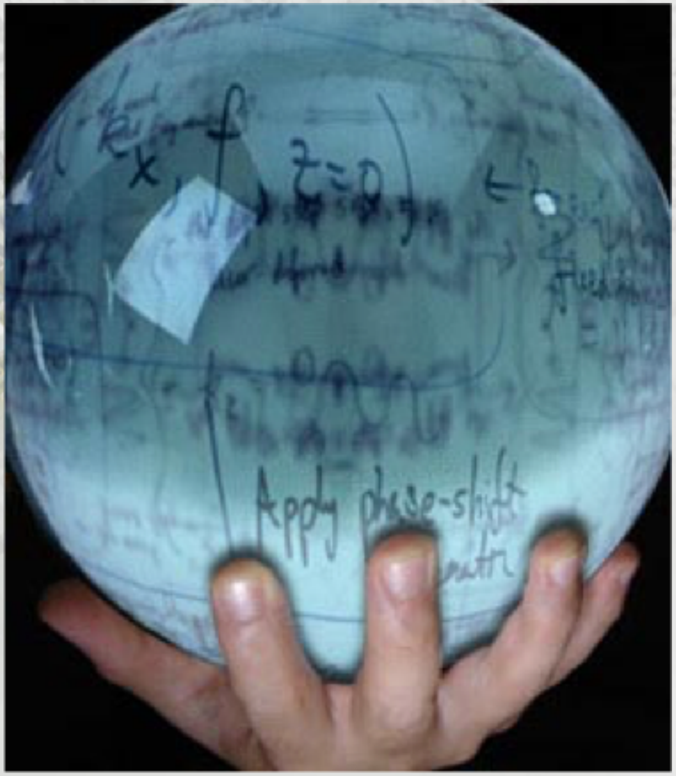


# Broad kinds of parsers

- Parsers for arbitrary grammars
  - Earley's method, CYK method
  - Usually, not used in practice (though might change)
- Top-down parsers
  - Construct parse tree in a top-down matter
  - Find the leftmost derivation
- Bottom-up parsers
  - Construct parse tree in a bottom-up manner
  - Find the rightmost derivation in a reverse order

# Top-Down Parsing: Predictive parsing

- Recursive descent
- LL(k) grammars



# Predictive parsing

- Given a grammar G and a word w attempt to derive w using G
- Idea
  - Apply production to leftmost nonterminal
  - Pick production rule based on next input token
- General grammar
  - More than one option for choosing the next production based on a token
- Restricted grammars (LL)
  - Know exactly which single rule to apply
  - May require some lookahead to decide

# Recursive descent parsing

- Define a function for every nonterminal
- Every function work as follows
  - Find applicable production rule
  - Terminal function checks match with next input token
  - Nonterminal function calls (recursively) other functions
- If there are several applicable productions for a nonterminal, use lookahead

# LL(k) grammars

- A grammar is in the class LL(K) when it can be derived via:
  - Top-down derivation
  - Scanning the input from left to right (L)
  - Producing the leftmost derivation (L)
  - With lookahead of k tokens (k)
- A language is said to be LL(k) when it has an LL(k) grammar

# FIRST sets

- FIRST(X) = { t | X $\rightarrow^*$ t β} ∪{Ɛ | X $\rightarrow^*$ Ɛ}
  - FIRST(X) = all terminals that α can appear as first in some derivation for X
    - + Ɛ if can be derived from X

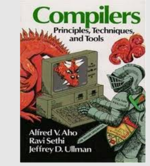
- Example:
  - FIRST( LIT ) = { true, false }
  - FIRST( ( E OP E ) ) = { ‘(‘ }
  - FIRST( not E ) = { not }

# FIRST sets

- No intersection between FIRST sets => can always pick a single rule
- If the FIRST sets intersect, may need longer lookahead
  - LL(k) = class of grammars in which production rule can be determined using a lookahead of k tokens
  - LL(1) is an important and useful class

# LL(1) grammars

- A grammar is in the class LL(K) iff
  - For every two productions $A \rightarrow \alpha$ and $A \rightarrow \beta$ we have
    - $FIRST(\alpha) \cap FIRST(\beta) = \{\}$ // including $\varepsilon$
    - If $\varepsilon \in FIRST(\alpha)$ then $FIRST(\beta) \cap FOLLOW(A) = \{\}$
    - If $\varepsilon \in FIRST(\beta)$ then $FIRST(\alpha) \cap FOLLOW(A) = \{\}$




p. 189

# FOLLOW sets

- What do we do with nullable ($\varepsilon$) productions?
  - $A \rightarrow B\ C\ D \quad B \rightarrow \varepsilon\ C \rightarrow \varepsilon$
  - Use what comes afterwards to predict the right production
- For every production rule $A \rightarrow \alpha$
  - FOLLOW(A) = set of tokens that can immediately follow A
- Can predict the alternative $A_k$ for a non-terminal N when the lookahead token is in the set
  - FIRST($A_k$) $\rightarrow$ (if $A_k$ is nullable then FOLLOW(N))

# FOLLOW sets: Constraints

- $\$ \in$ FOLLOW(S)
- FIRST(β) – {Ɛ} ⊆ FOLLOW(X)
  - For each A → α X β
- FOLLOW(A) ⊆ FOLLOW(X)
  - For each A → α X β and Ɛ ∈ FIRST(β)

# Prediction Table

- A → α

- T[A,t] = α    if  t ∈FIRST(α)
- T[A,t] = α    if  Ɛ ∈ FIRST(α) and t ∈ FOLLOW(A)
  - t can also be $

- T is not well defined ➔ the grammar is not LL(1)

# Problem 1: productions with common prefix

$term \rightarrow$ ID | indexed_elem
$indexed\_elem \rightarrow$ ID [ expr ]

- FIRST(term) = { ID }
- FIRST(indexed_elem) = { ID }

- FIRST/FIRST conflict

# Solution: left factoring

- Rewrite the grammar to be in LL(1)

term → ID | indexed_elem
indexed_elem → ID [ expr ]

⇩

term → ID after_ID
After_ID → [ expr ] | ε

Intuition: just like factoring x*y + x*z into x*(y+z)

# Problem 2: null productions

$S \rightarrow A\ a\ b$
$A \rightarrow a \mid \varepsilon$

- FIRST(S) = { a }    FOLLOW(S) = { }
- FIRST(A) = { a , ε }    FOLLOW(A) = { a }

- FIRST/FOLLOW conflict

# Solution: substitution

S → A a b
A → a | ε

Substitute A in S

S → a a b | a b

Left factoring

S → a after_A
after_A → a b | b

# Problem 3: left recursion

$E \rightarrow E$ - term | term

- Left recursion cannot be handled with a bounded lookahead

- What can we do?


p. 130

# Left recursion removal

$N \rightarrow N\alpha \mid \beta$

$G_1$

$N \rightarrow \beta N'$
$N' \rightarrow \alpha N' \mid \varepsilon$

$G_2$

Can be done algorithmically.
Problem: grammar becomes mangled beyond recognition

- $L(G_1) = \beta, \beta\alpha, \beta\alpha\alpha, \beta\alpha\alpha\alpha, \ldots$
- $L(G_2)$ = same

- For our 3rd example:

E → E - term | term

E → term TE | term
TE → - term TE | ε

# Bottom-up parsing

# Bottom-up parsing: LR(k) Grammars

- A grammar is in the class LR(K) when it can be derived via:
  - Bottom-up derivation
  - Scanning the input from left to right (L)
  - Producing the rightmost derivation (R)
  - With lookahead of k tokens (k)
- A language is said to be LR(k) if it has an LR(k) grammar
- The simplest case is LR(0), which we will discuss

# Terminology: Reductions & Handles

- The opposite of derivation is called *reduction*
  - Let A ➔ α be a production rule
  - Derivation: βAμ ➔ βαμ
  - Reduction: βαμ ➔ βAμ

- A *handle* is the reduced substring
  - α is the handles for βαμ

# How does the parser know what to do?

- A state will keep the info gathered on handle(s)
  - A state in the "control" of the PDA
  - Also (part of) the stack alpha bet

Set of LR(0) items

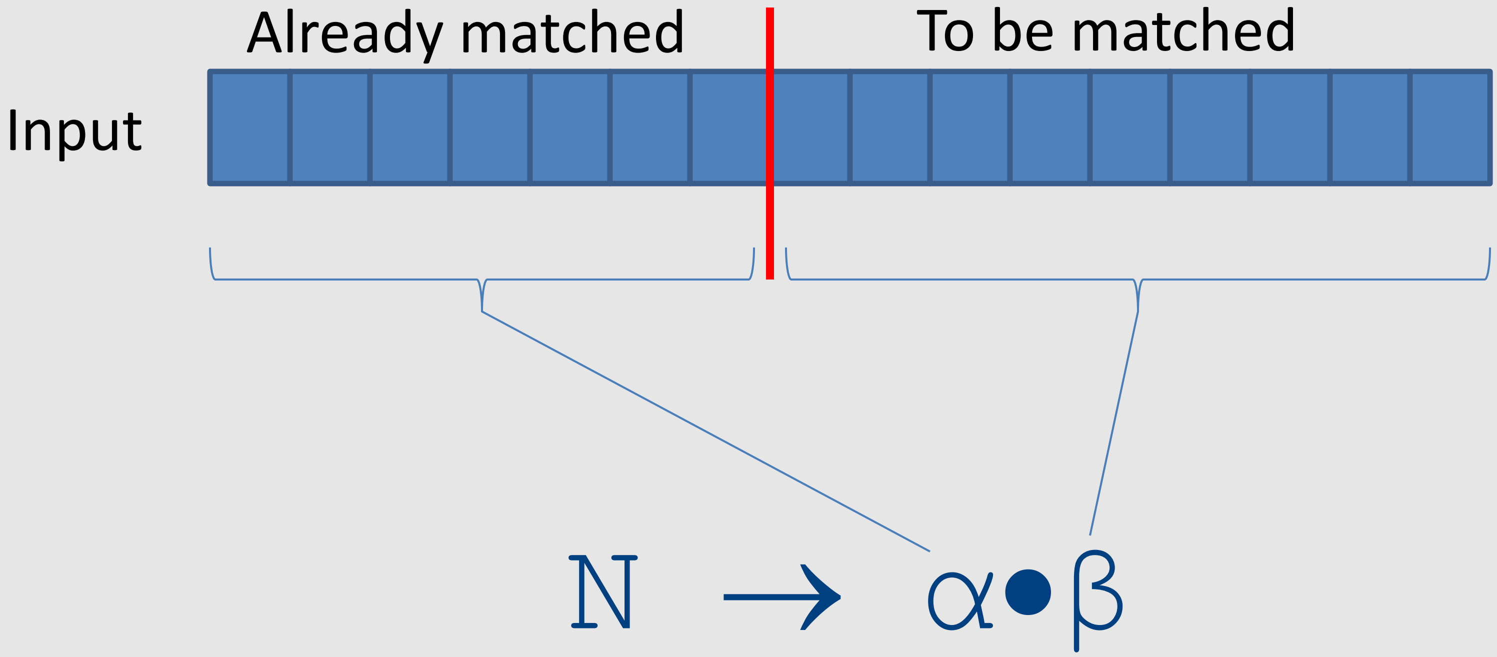
- A table will tell it "what to do" based on current state and next token
  - The transition function of the PDA

- A stack will records the "nesting level"
  - Prefixes of handles

# Constructing an LR parsing table

- Construct a (determinized) transition diagram from LR items
- If there are conflicts – stop
- Fill table entries from diagram

# LR item



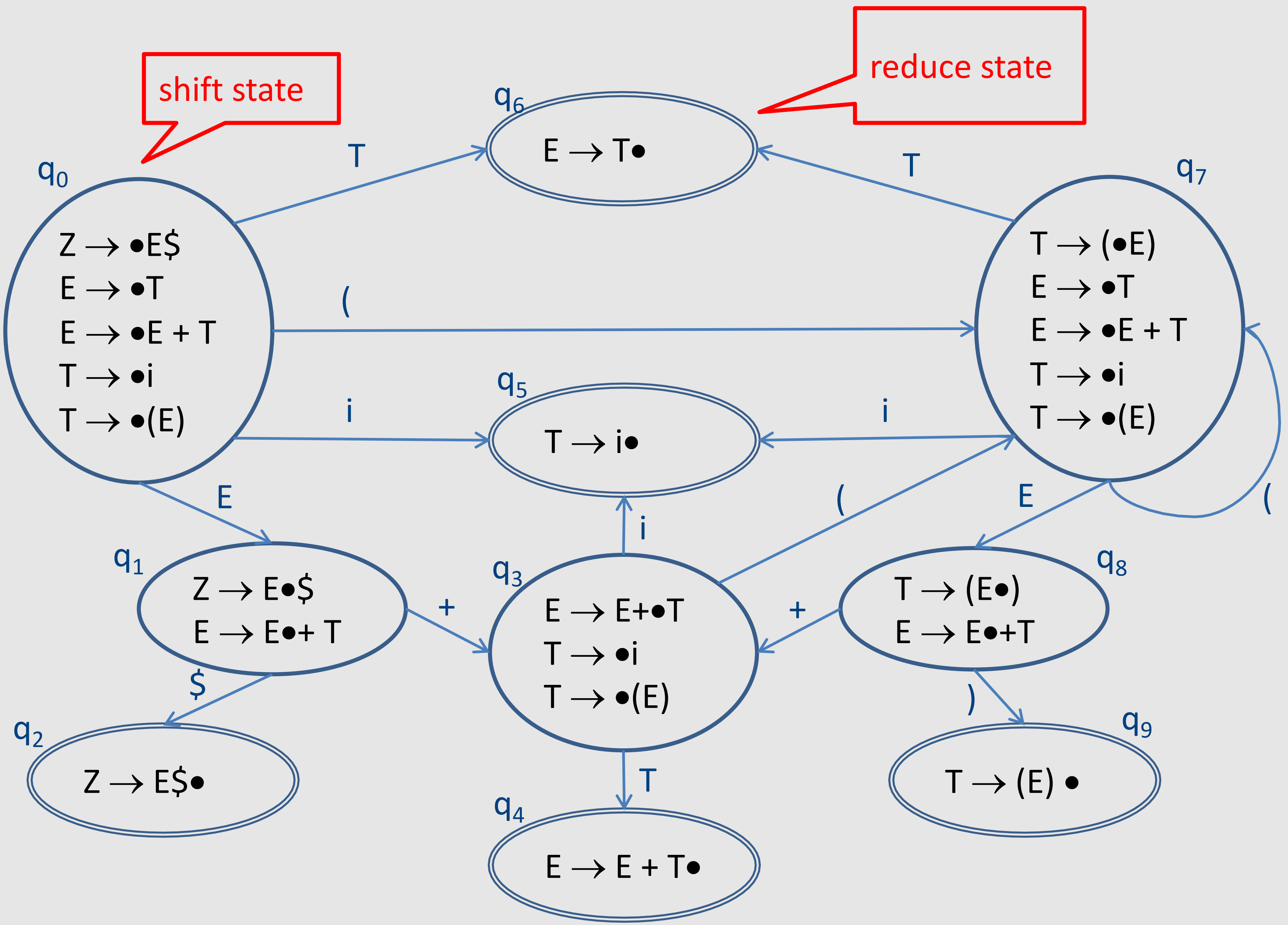

Hypothesis about αβ being a possible handle, so far we've matched α, expecting to see β

# Types of LR(0) items

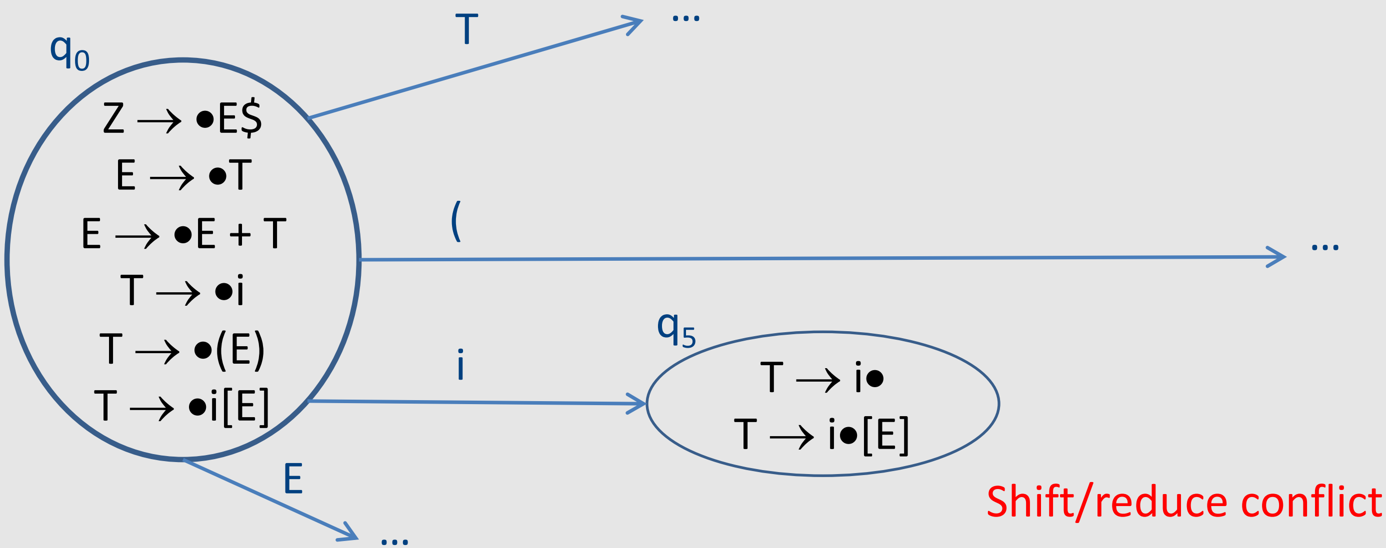
$N \rightarrow \alpha \bullet \beta$ Shift Item

$N \rightarrow \alpha \beta \bullet$ Reduce Item

# LR(0) automaton example

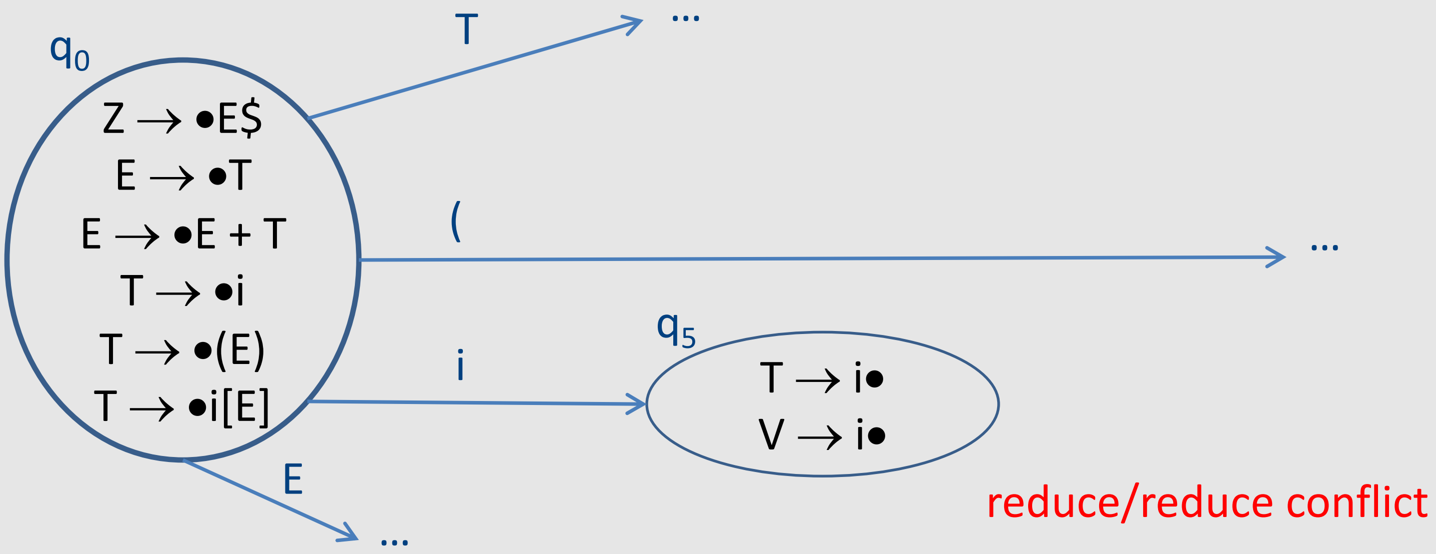


# LR(0) conflicts




Z → E $
E → T
E → E + T
T → i
T → ( E )
T → i[E]

# LR(0) conflicts




Z → E $
E → T
E → E + T
T → i
V → i
T → ( E )

# LR(0) conflicts

- Any grammar with an $\varepsilon$-rule cannot be LR(0)
- Inherent shift/reduce conflict
  - $A \rightarrow \varepsilon\bullet$ – reduce item
  - $P \rightarrow \alpha\bullet A\beta$ – shift item
  - $A \rightarrow \varepsilon\bullet$ can always be predicted from $P \rightarrow \alpha\bullet A\beta$

# LR variants

- LR(0) – what we've seen so far
- SLR
  - Removes infeasible reduce actions via FOLLOW set reasoning
- LR(1)
  - LR(0) with one lookahead token in items
- LALR(0)
  - LR(1) with merging of states with same LR(0) component

# Semantic Analysis

# Abstract Syntax Tree

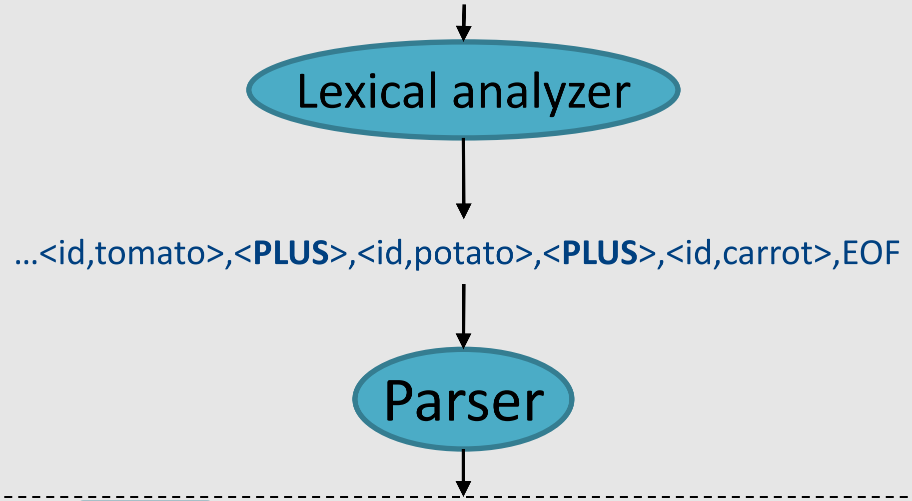
- AST is a simplification of the parse tree
- Can be built by traversing the parse tree
  - E.g., using visitors
- Can be built directly during parsing
  - Add an action to perform on each production rule
  - Similarly to the way a parse tree is constructed

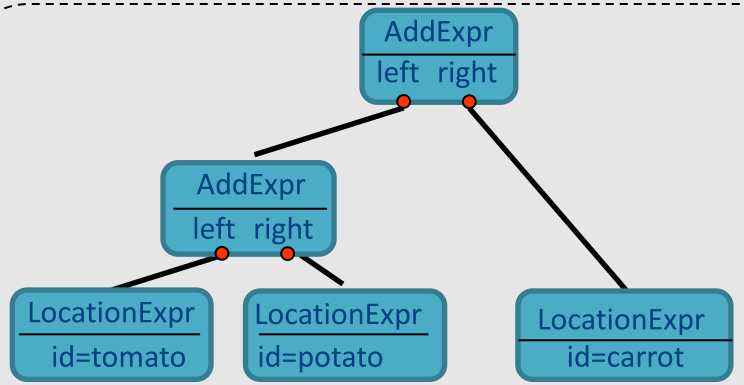
# Abstract Syntax Tree

- The interface between the parser and the rest of the compiler
  - Separation of concerns
  - Reusable, modular and extensible
- The AST is defined by a context free grammar
  - The grammar of the AST can be ambiguous!
    - $E \rightarrow E + E$
    - Is this a problem?
- Keep syntactic information
  - Why?

# What we want

Potato potato;
Carrot carrot;
x = tomato + potato + carrot




| symbol | kind | type | properties |
|---|---|---|---|
| x | var | ? | |
| tomato | var | ? | |
| potato | var | Potato | |
| carrot | var | Carrot | |

‘tomato’ is undefined

‘potato’ used before initialized

Cannot add Potato and Carrot

# Context Analysis

- Check properties contexts of in which constructs occur
  - Properties that cannot be formulated via CFG
    - Type checking
    - Declare before use
      - Identifying the same word “w” re-appearing – wbw
    - Initialization
    - …
  - Properties that are hard to formulate via CFG
    - “break” only appears inside a loop
    - …
- Processing of the AST

# Context Analysis

- Identification
  - Gather information about each named item in the program
  - e.g., what is the declaration for each usage
- Context checking
  - Type checking
  - e.g., the condition in an if-statement is a Boolean

# Scopes

- Typically stack structured scopes

- Scope entry
  - push new empty scope element
- Scope exit
  - pop scope element and discard its content
- Identifier declaration
  - identifier created inside top scope
- Identifier Lookup
  - Search for identifier top-down in scope stack

# Scope and symbol table

- Scope x Identifier -> properties
  - Expensive lookup
- A better solution
  - hash table over identifiers

# Types

- What is a type?
  - Simplest answer: a set of values + allowed operations
  - Integers, real numbers, booleans, …
- Why do we care?
  - Code generation: $1 := $1 + $2
  - Safety
    - Guarantee that certain errors cannot occur at runtime
  - Abstraction
    - Hide implementation details
  - Documentation
  - Optimization

# Typing Rules

If E1 has type int and E2 has type int,
then E1 + E2 has type int

$$\frac{E1 : int \qquad E2 : int}{E1 + E2 : int}$$

# Syntax Directed Translation

- Semantic attributes
  - Attributes attached to grammar symbols
- Semantic actions
  - How to update the attributes

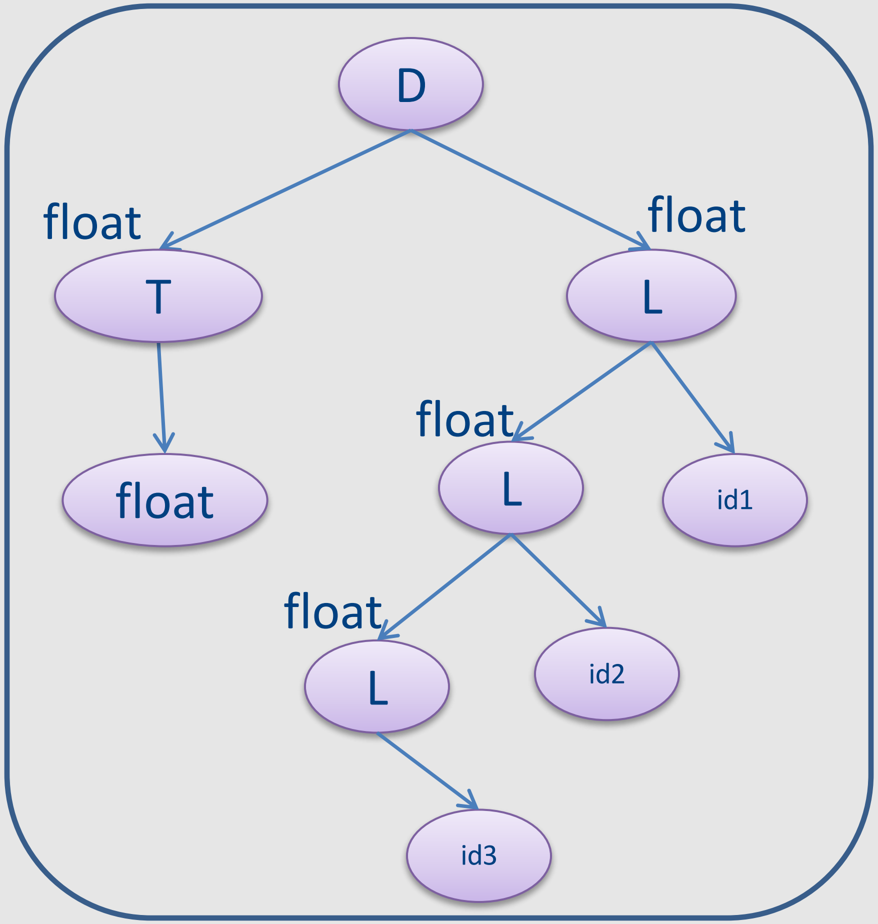
- Attribute grammars

# Attribute grammars

- Attributes
  - Every grammar symbol has attached attributes
    - Example: Expr.type
- Semantic actions
  - Every production rule can define how to assign values to attributes
    - Example:
      Expr → Expr + Term
      Expr.type = Expr1.type when (Expr1.type == Term.type)
      Error otherwise

# Example

float x,y,z




| Production | Semantic Rule |
|---|---|
| D → T L | L.in = T.type |
| T → int | T.type = integer |
| T → float | T.type = float |
| L → L1, id | L1.in = L.in<br>addType(id.entry,L.in) |
| L → id | addType(id.entry,L.in) |

# Attribute Evaluation

- Build the AST
- Fill attributes of terminals with values derived from their representation
- Execute evaluation rules of the nodes to assign values until no new values can be assigned
  - In the right order such that
    - No attribute value is used before its available
    - Each attribute will get a value only once

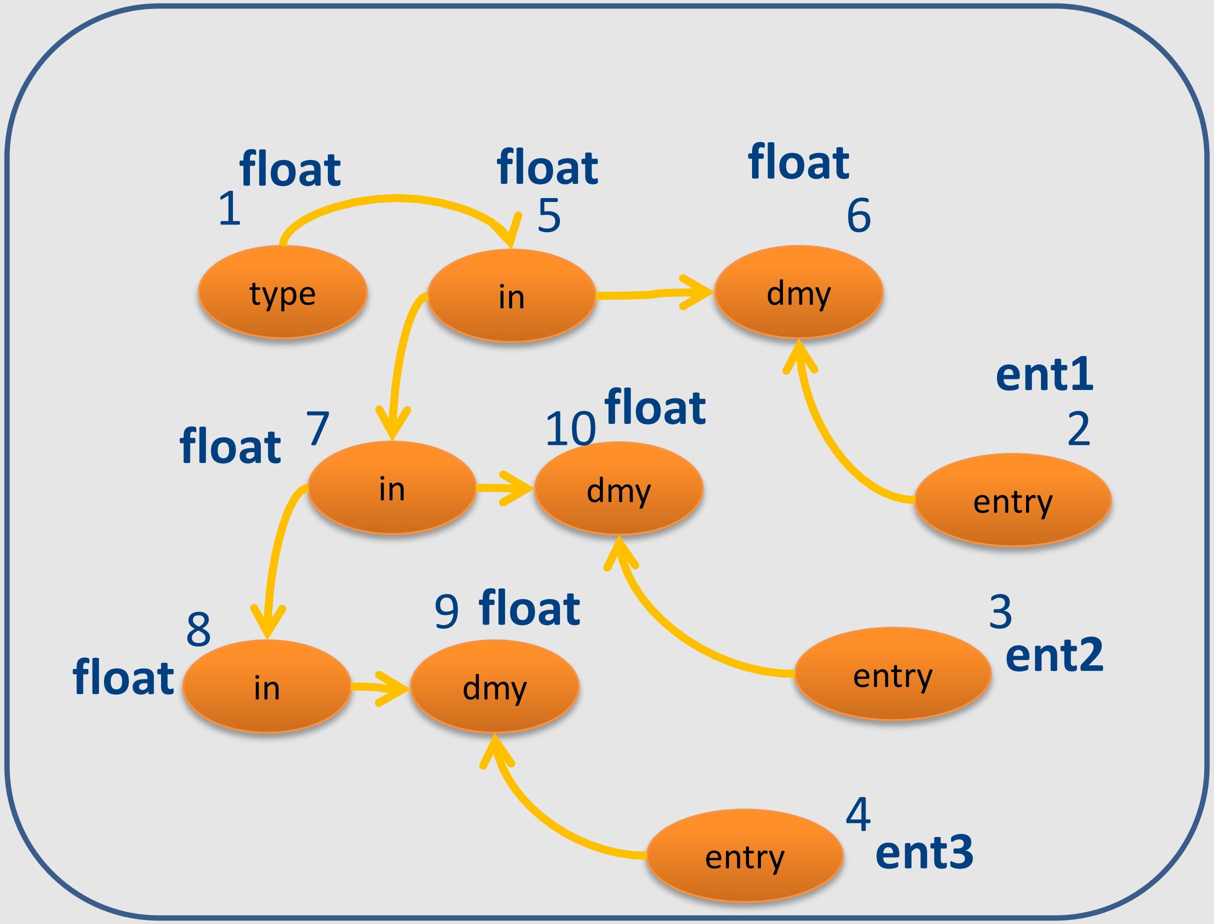
# Dependencies

- A semantic equation a = b1,…,bm requires computation of b1,…,bm to determine the value of a

- The value of a depends on b1,…,bm
  - We write a → bi

# Example

float x,y,z



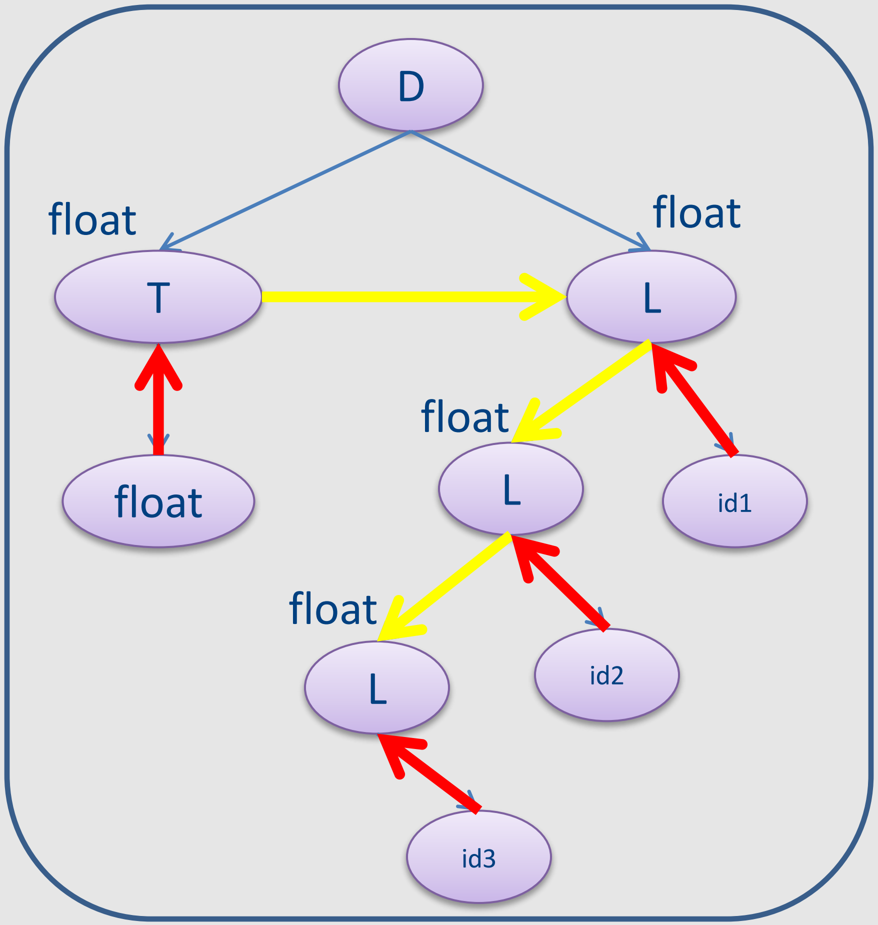
# Inherited vs. Synthesized Attributes

- Synthesized attributes
  - Computed from children of a node
- Inherited attributes
  - Computed from parents and siblings of a node

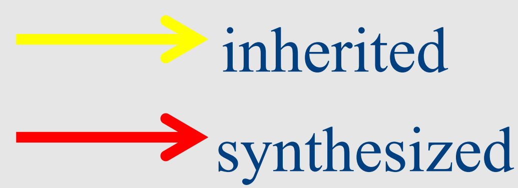
- Attributes of tokens are technically considered as synthesized attributes

# example

float x,y,z




| Production | Semantic Rule |
|---|---|
| D → T L | L.in = T.type |
| T → int | T.type = integer |
| T → float | T.type = float |
| L → L1, id | L1.in = L.in<br>addType(id.entry,L.in) |
| L → id | addType(id.entry,L.in) |



# S-attributed Grammars

- Special class of attribute grammars
- Only uses synthesized attributes (S-attributed)
- No use of inherited attributes

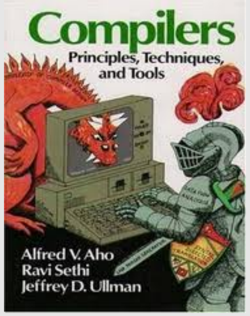
- Can be computed by any bottom-up parser during parsing
- Attributes can be stored on the parsing stack
- Reduce operation computes the (synthesized) attribute from attributes of children

# L-attributed grammars

- L-attributed attribute grammar when every attribute in a production A → X1…Xn is
  - A synthesized attribute, or
  - An inherited attribute of Xj, 1 <= j <=n that only depends on
    - Attributes of X1…Xj-1 to the left of Xj, or
    - Inherited attributes of A

# Intermediate Representation

# **T**hree-**A**ddress **C**ode IR




Chapter 8

- A popular form of IR
- High-level assembly where instructions have at most three operands

# Variable assignments

- var = constant ;
- $var_1$ = $var_2$ ;
- $var_1$ = $var_2$ **op** $var_3$ ;
- $var_1$ = constant **op** $var_2$ ;
- $var_1$ = $var_2$ **op** constant ;
- var = $constant_1$ **op** $constant_2$ ;
- Permitted operators are **+, -, *, /, %**

In the impl. var is replaced by a pointer to the symbol table

A compiler-generated temporary can be used instead of a var

# Control flow instructions

- Label introduction

  ```
  _label_name:
  ```

  Indicates a point in the code that can be jumped to
- Unconditional jump: go to instruction following label L

  ```
  Goto L;
  ```
- Conditional jump: test condition variable t;
  if 0, jump to label L

  ```
  IfZ t Goto L;
  ```
- Similarly : test condition variable t;
  if not zero, jump to label L

  ```
  IfNZ t Goto L;
  ```

# Procedures / Functions

- A procedure call instruction **pushes** arguments to stack and **jumps** to the function label
  A statement `x=f(a1,…,an);` looks like

```
Push a1; … Push an;
Call f;
Pop x; // pop returned value, and copy to it
```

- Returning a value is done by **pushing** it to the stack (`return x;`)

```
Push x;
```

- **Return control** to caller (and **roll up stack**)

```
Return;
```

# TAC generation

- At this stage in compilation, we have
  - an AST
  - annotated with scope information
  - and annotated with type information
- To generate TAC for the program, we do recursive tree traversal
  - Generate TAC for any subexpressions or substatements
  - Using the result, generate TAC for the overall expression

# **cgen** for binary operators

**cgen**($e_1 + e_2$) = {
Choose a new temporary *t*
Let $t_1$ = **cgen**($e_1$)
Let $t_2$ = **cgen**($e_2$)
Emit( $t = t_1 + t_2$ )
Return *t*
}

# **cgen** for `if-then-else`

**cgen**(if (e) $s_1$ else $s_2$)

Let _t = **cgen**(e)

Let $L_{true}$ be a new label

Let $L_{false}$ be a new label

Let $L_{after}$ be a new label

Emit( IfZ _t Goto $L_{false}$; )

**cgen**($s_1$)

Emit( Goto $L_{after}$; )

Emit( $L_{false}$: )

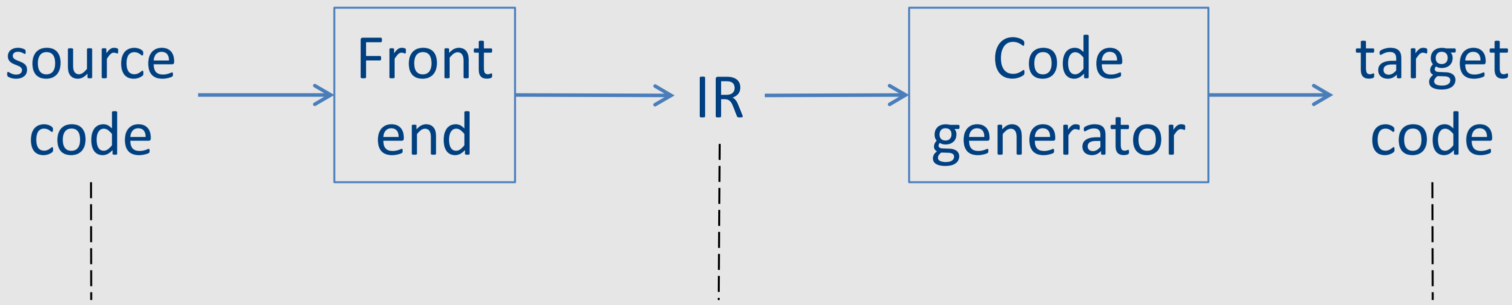
**cgen**($s_2$)

Emit( Goto $L_{after}$;)

Emit( $L_{after}$: )

# IR Optimization

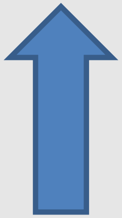
# Optimization points




**User**
profile program
change algorithm

**Compiler**
intraprocedural IR
Interprocedural IR
**IR optimizations**

**Compiler**
register allocation
instruction selection
peephole transformations



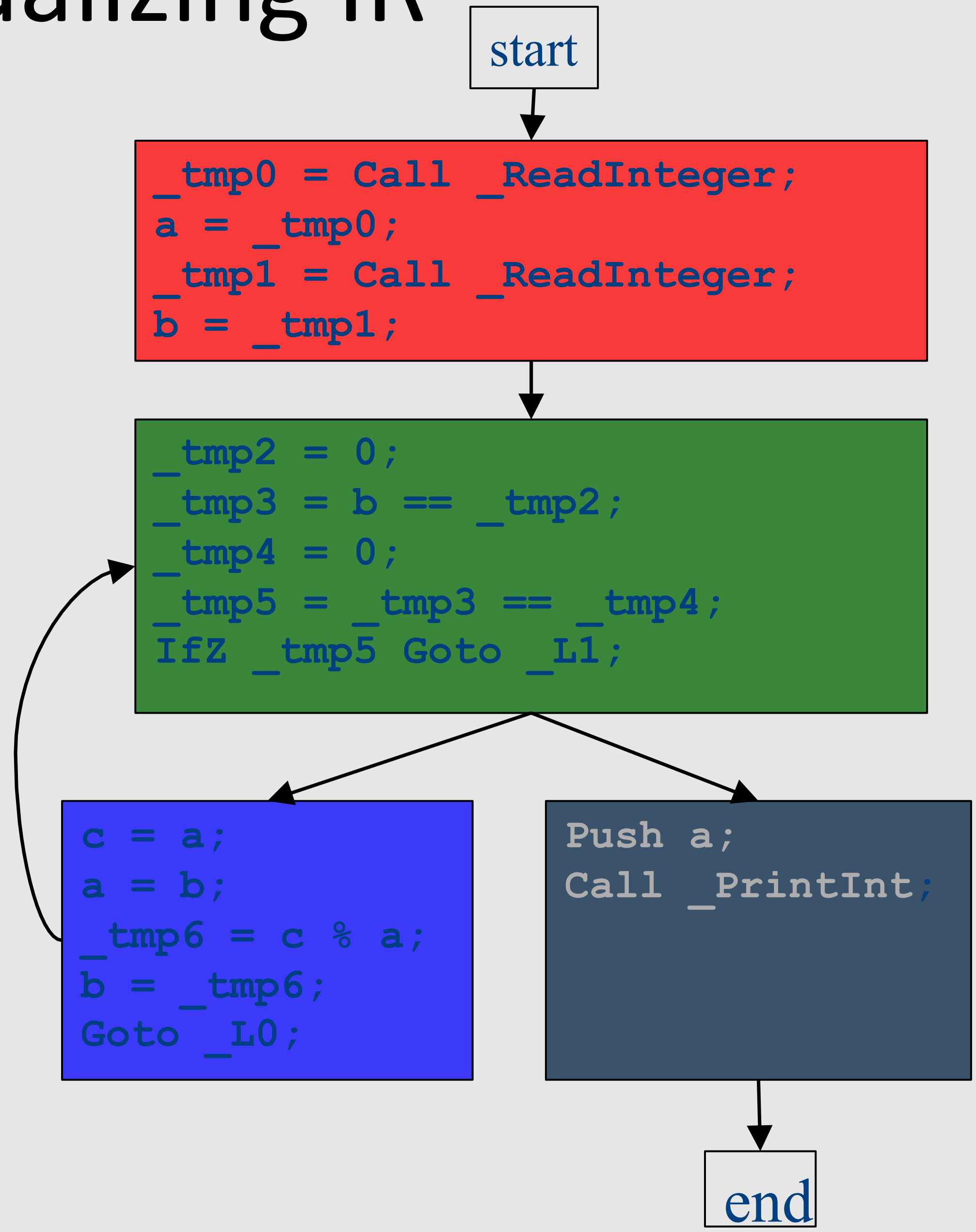
**now**

# Overview of IR optimization

- Formalisms and Terminology
  - Control-flow graphs
  - Basic blocks
- Local optimizations
  - Speeding up small pieces of a procedure
- Global optimizations
  - Speeding up procedure as a whole
- The dataflow framework
  - Defining and implementing a wide class of optimizations

# Visualizing IR

```
main:
    _tmp0 = Call _ReadInteger;
    a = _tmp0;
    _tmp1 = Call _ReadInteger;
    b = _tmp1;
_L0:
    _tmp2 = 0;
    _tmp3 = b == _tmp2;
    _tmp4 = 0;
    _tmp5 = _tmp3 == _tmp4;
    IfZ _tmp5 Goto _L1;
    c = a;
    a = b;
    _tmp6 = c % a;
    b = _tmp6;
    Goto _L0;
_L1:
    Push a;
    Call _PrintInt;
```



# Control-Flow Graphs

- A control-flow graph (CFG) is a graph of the basic blocks in a function
- The term CFG is overloaded – from here on out, we'll mean “control-flow graph” and not “context free grammar”
- Each edge from one basic block to another indicates that control can flow from the end of the first block to the start of the second block
- There is a dedicated node for the start and end of a function

# **C**ommon **S**ubexpression **E**limination

- If we have two variable assignments
  v1 = a op b
  …
  v2 = a op b
- and the values of v1, a, and b have not changed between the assignments, rewrite the code as
  v1 = a op b
  …
  v2 = v1
- Eliminates useless recalculation
- Paves the way for later optimizations

# Common Subexpression Elimination

- If we have two variable assignments
  v1 = a op b [or: v1 = a]
  …
  v2 = a op b [or: v2 = a]
- and the values of v1, a, and b have not changed between the assignments, rewrite the code as
  v1 = a op b [or: v1 = a]
  …
  v2 = v1
- Eliminates useless recalculation
- Paves the way for later optimizations

# Copy Propagation

- If we have a variable assignment
  v1 = v2
  then as long as v1 and v2 are not reassigned, we can rewrite expressions of the form
  a = … v1 …
  as
  a = … v2 …
  provided that such a rewrite is legal

# Dead Code Elimination

- An assignment to a variable v is called dead if the value of that assignment is never read anywhere
- Dead code elimination removes dead assignments from IR
- Determining whether an assignment is dead depends on what variable is being assigned to and when it's being assigned

# Live variables

- The analysis corresponding to dead code elimination is called liveness analysis
- A variable is live at a point in a program if later in the program its value will be read before it is written to again
- Dead code elimination works by computing liveness for each variable, then eliminating assignments to dead variables

# Local vs. global optimizations

- An optimization is local if it works on just a single basic block
- An optimization is global if it works on an entire control-flow graph of a procedure
- An optimization is interprocedural if it works across the control-flow graphs of multiple procedure
  - We won't talk about this in this course

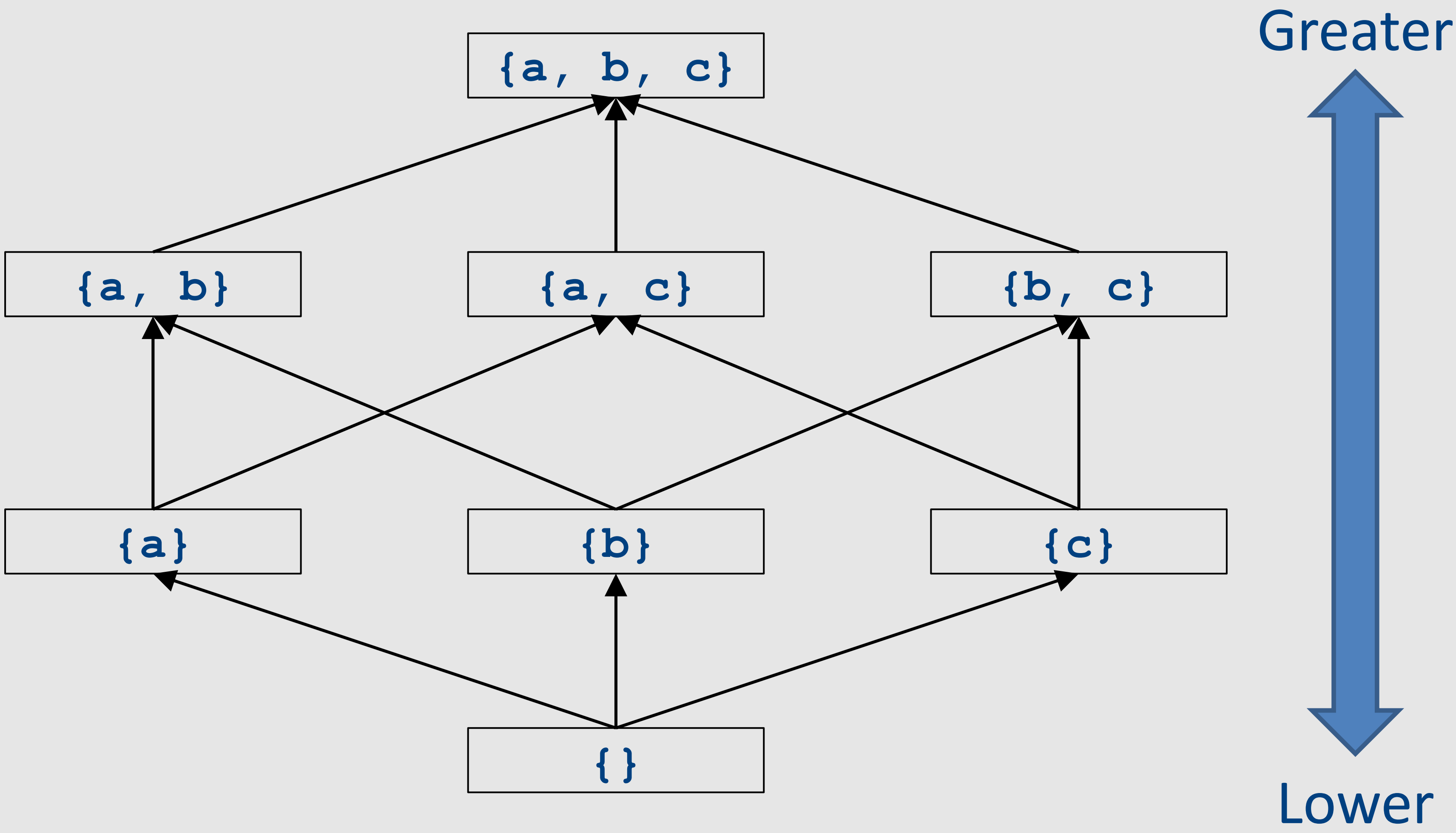
# Abstract Interpretation

- Theoretical foundations of program analysis

- Cousot and Cousot 1977

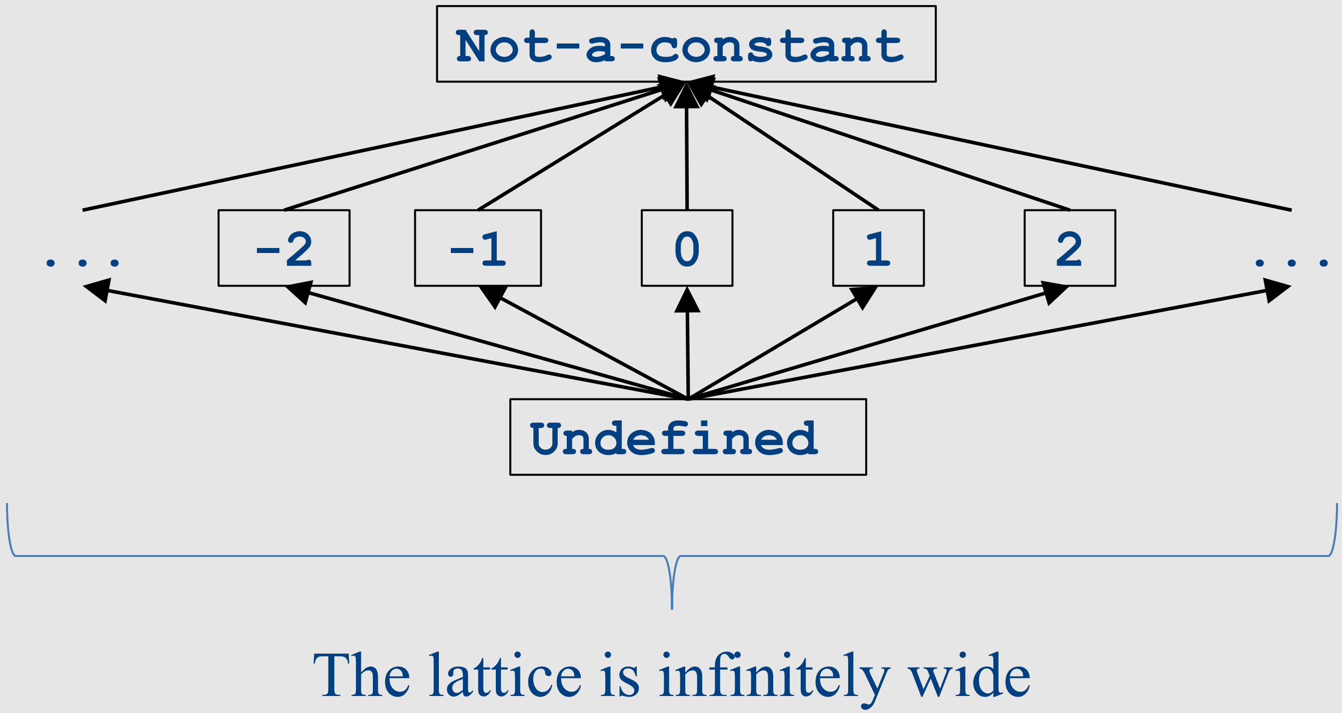
- Abstract meaning of programs
  - Executed at compile time

# Join semilattices and ordering



# A semilattice for constant propagation

- One possible semilattice for this analysis is shown here (for each variable):



# Monotone transfer functions

- A transfer function *f* is monotone iff

  if $x \sqsubseteq y$, then $f(x) \sqsubseteq f(y)$
- Intuitively, if you know less information about a program point, you can't “gain back” more information about that program point
- Many transfer functions are monotone, including those for liveness and constant propagation
- Note: Monotonicity does **not** mean that $x \sqsubseteq f(x)$
  - (This is a different property called extensivity)

# The grand result

- Theorem: A dataflow analysis with a **finite-height semilattice** and family of **monotone transfer functions** always terminates
- Proof sketch:
  - The join operator can only bring values up
  - Transfer functions can never lower values back down below where they were in the past (monotonicity)
  - Values cannot increase indefinitely (finite height)

# Code Generation

# From TAC IR to Assembly

- Shown in project & recitation

# Code generation for procedure calls

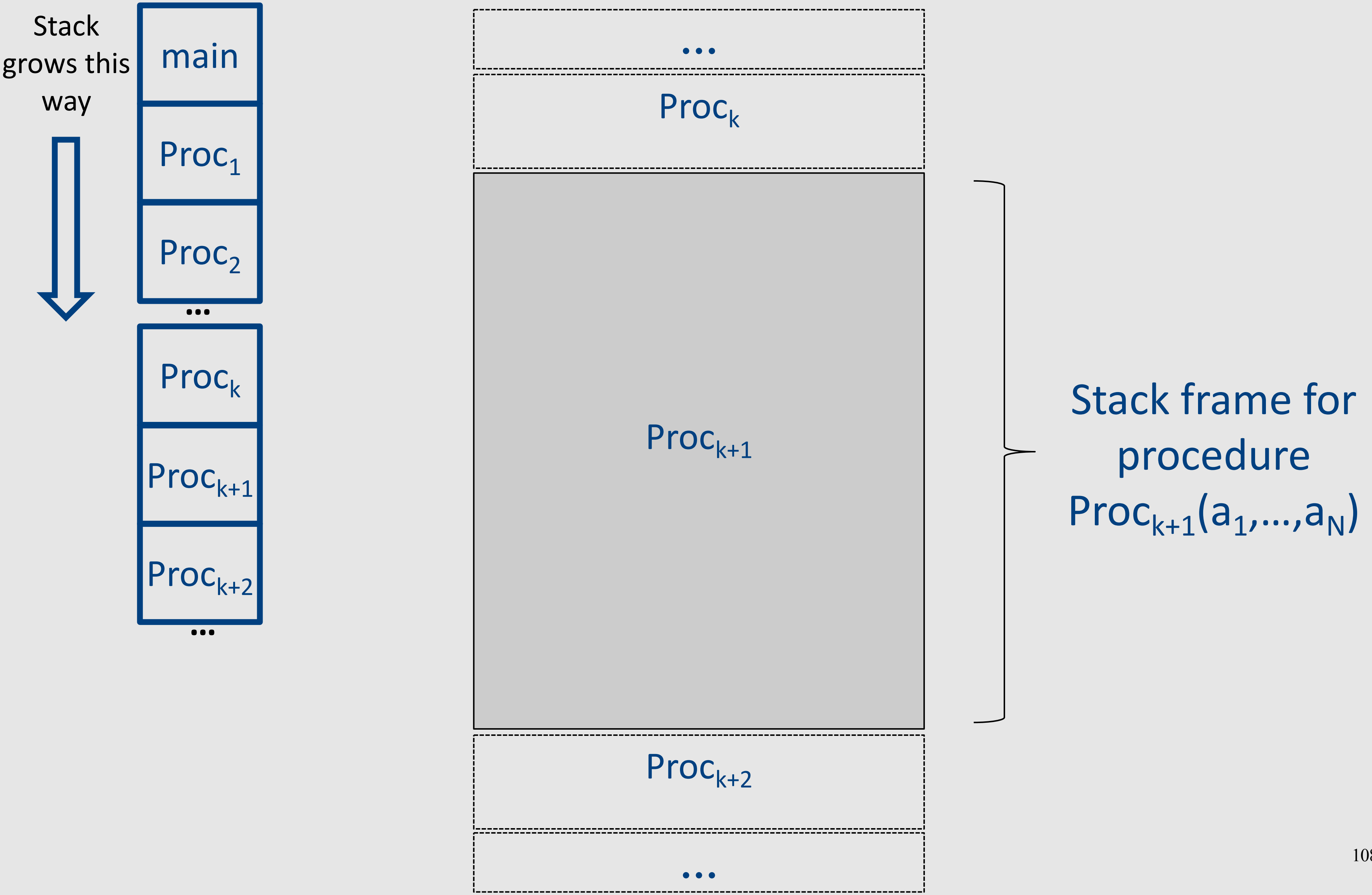
- Compile time generation of code for procedure invocations

- Activation Records (aka Stack Frames)

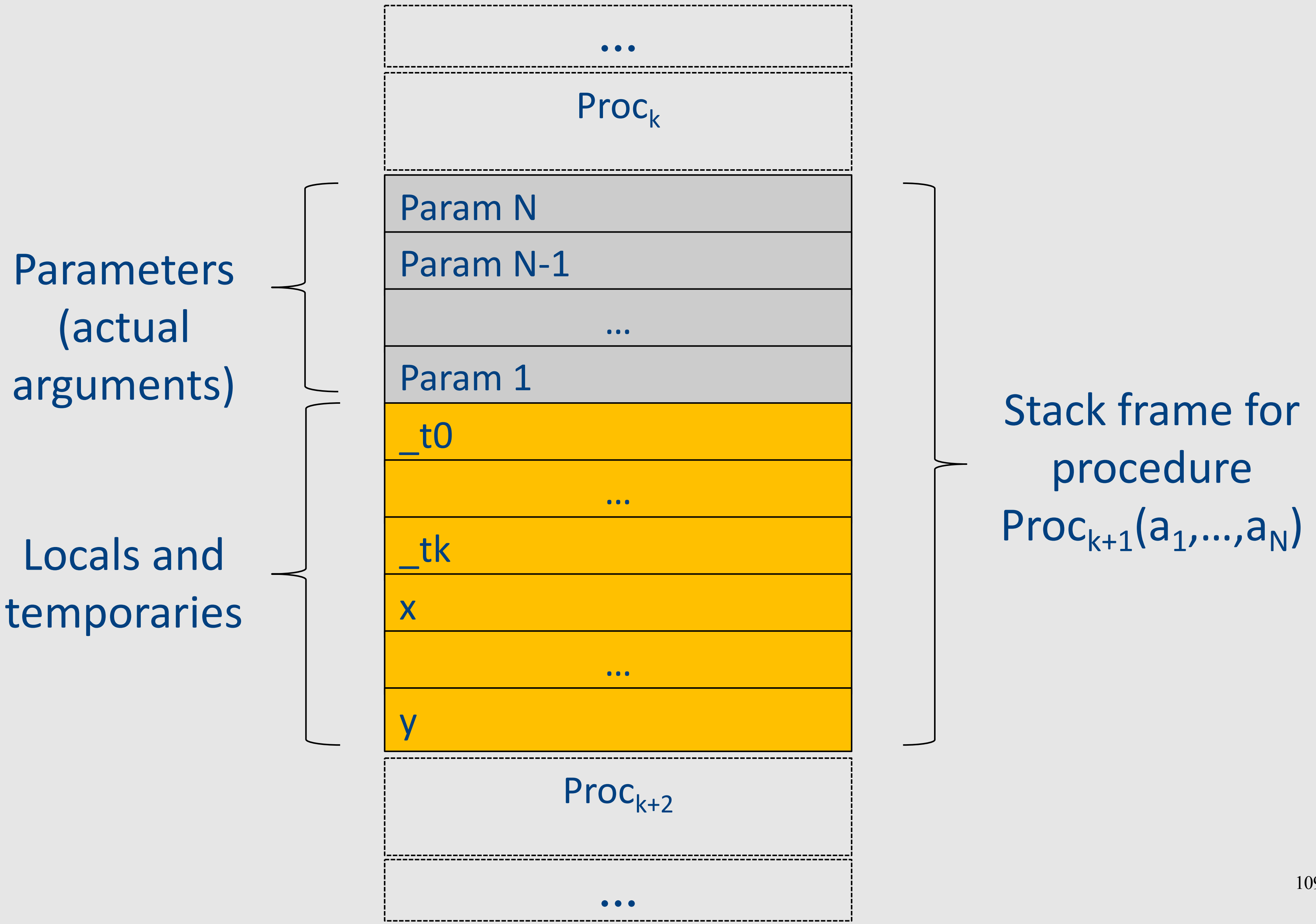
# Supporting Procedures

- **Stack**: a new computing environment
  - e.g., temporary memory for **local variables**
- Passing information into the new environment
  - **Parameters**
- **Transfer** of **control** to/from procedure
- Handling return values

# Abstract Activation Record Stack



# Abstract Stack Frame



# Static (lexical) Scoping

```
main ( )
{
    int a = 0 ;
    int b = 0 ;
    {
        int b = 1 ;
        {
            int a = 2 ;
            printf ("%d %d\n", a, b)
        }
        {
            int b = 3 ;
            printf ("%d %d\n", a, b) ;
        }
        printf ("%d %d\n", a, b) ;
    }
    printf ("%d %d\n", a, b) ;
}
```

$B_0$ $B_1$ $B_2$ $B_3$

a name refers to its (closest) enclosing scope

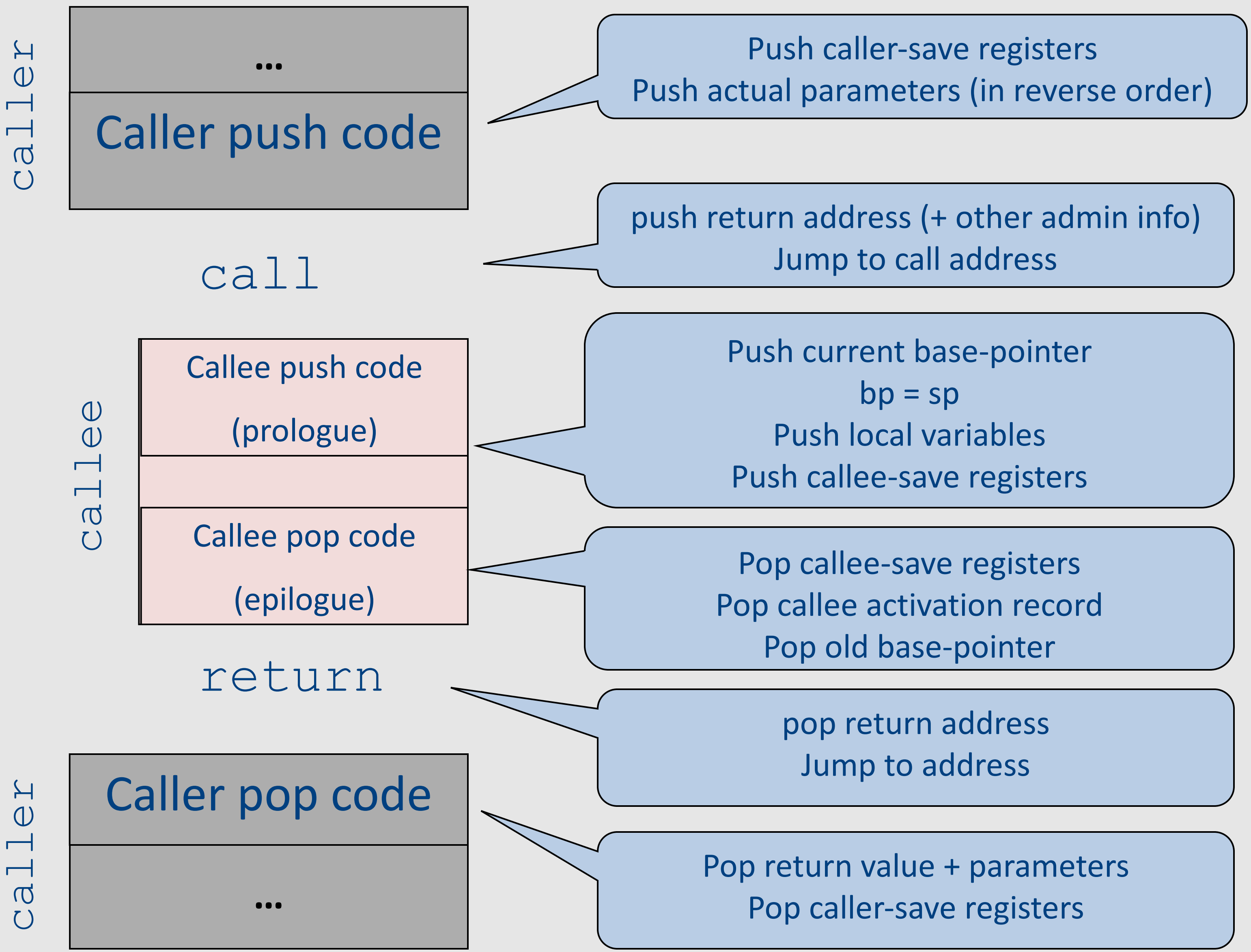
**known at compile time**

| Declaration | Scopes |
|---|---|
| a=0 | B0,B1,B3 |
| b=0 | B0 |
| b=1 | B1,B2 |
| a=2 | B2 |
| b=3 | B3 |

# Dynamic Scoping

- Each identifier is associated with a global stack of bindings
- When entering scope where identifier is declared
  - push declaration on identifier stack
- When exiting scope where identifier is declared
  - pop identifier stack
- **Evaluating the identifier in any context binds to the current top of stack**
- Determined **at runtime**

# Call Sequences



# “To Callee-save or to Caller-save?”

- Callee-saved registers need only be saved when callee modifies their value
- Some heuristics and conventions are followed

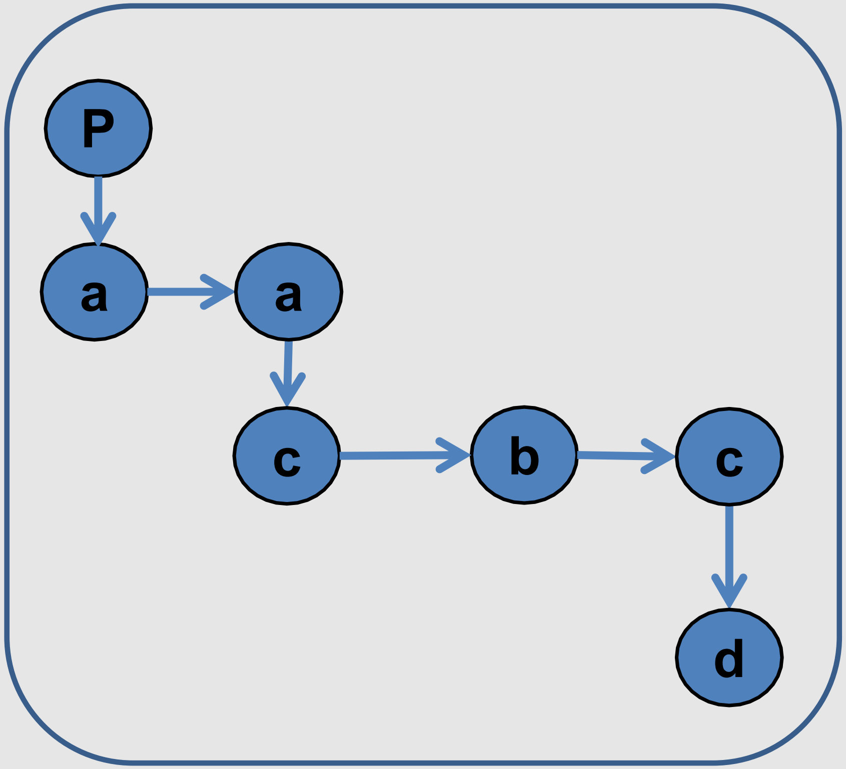
# Nested Procedures

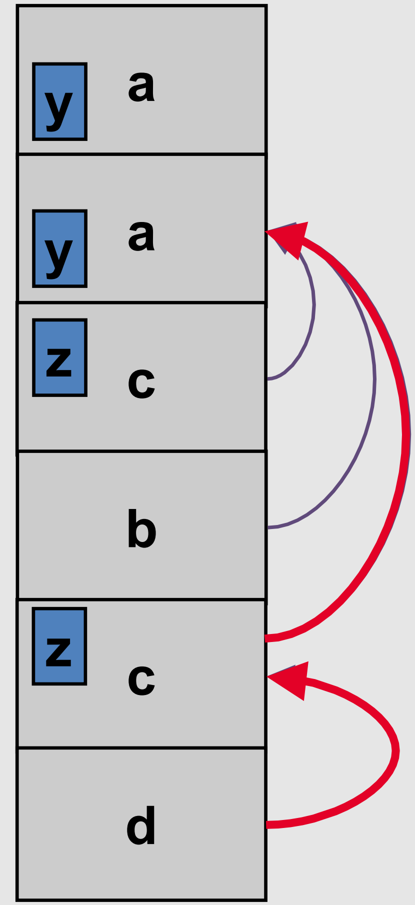
- problem: a routine may need to access variables of another routine that contains it statically
- solution: lexical pointer (a.k.a. access link) in the activation record
- lexical pointer points to the last activation record of the nesting level above it
  - in our example, lexical pointer of d points to activation records of c
- lexical pointers created at runtime
- number of links to be traversed is known at compile time

# Lexical Pointers

```
program p(){
   int x;
  procedure a(){
     int y;
   procedure b(){ c() };
     procedure c(){
        int z;
       procedure d(){
          y := x + z
       };
        … b() … d() …
     }
    … a() … c() …
  }
   a()
}
```

Possible call sequence:
p → a → a → c → b → c → d



# Register allocation

# Register allocation

- Number of registers is **limited**
- Need to **allocate** them in a clever way
  - Using registers intelligently is a critical step in any compiler
    - A good register allocator can generate code orders of magnitude better than a bad register allocator

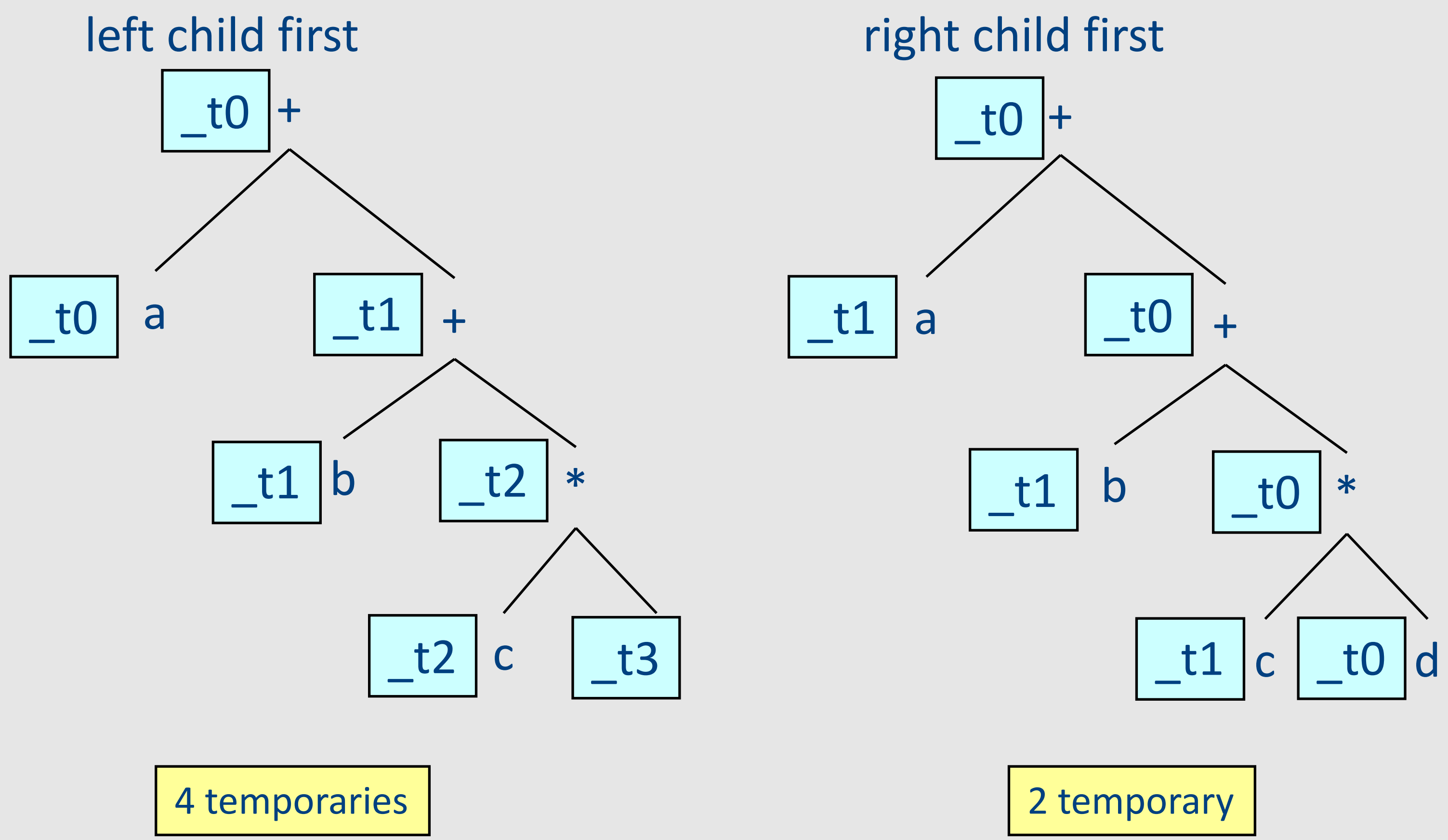
# Sethi-Ullman translation

- Algorithm by Ravi Sethi and Jeffrey D. Ullman to emit optimal TAC
  - Minimizes number of temporaries
- Main data structure in algorithm is a stack of temporaries
  - Stack corresponds to recursive invocations of _t = **cgen**(e)
  - All the temporaries on the stack are live
    - Live = contain a value that is needed later on

# Example

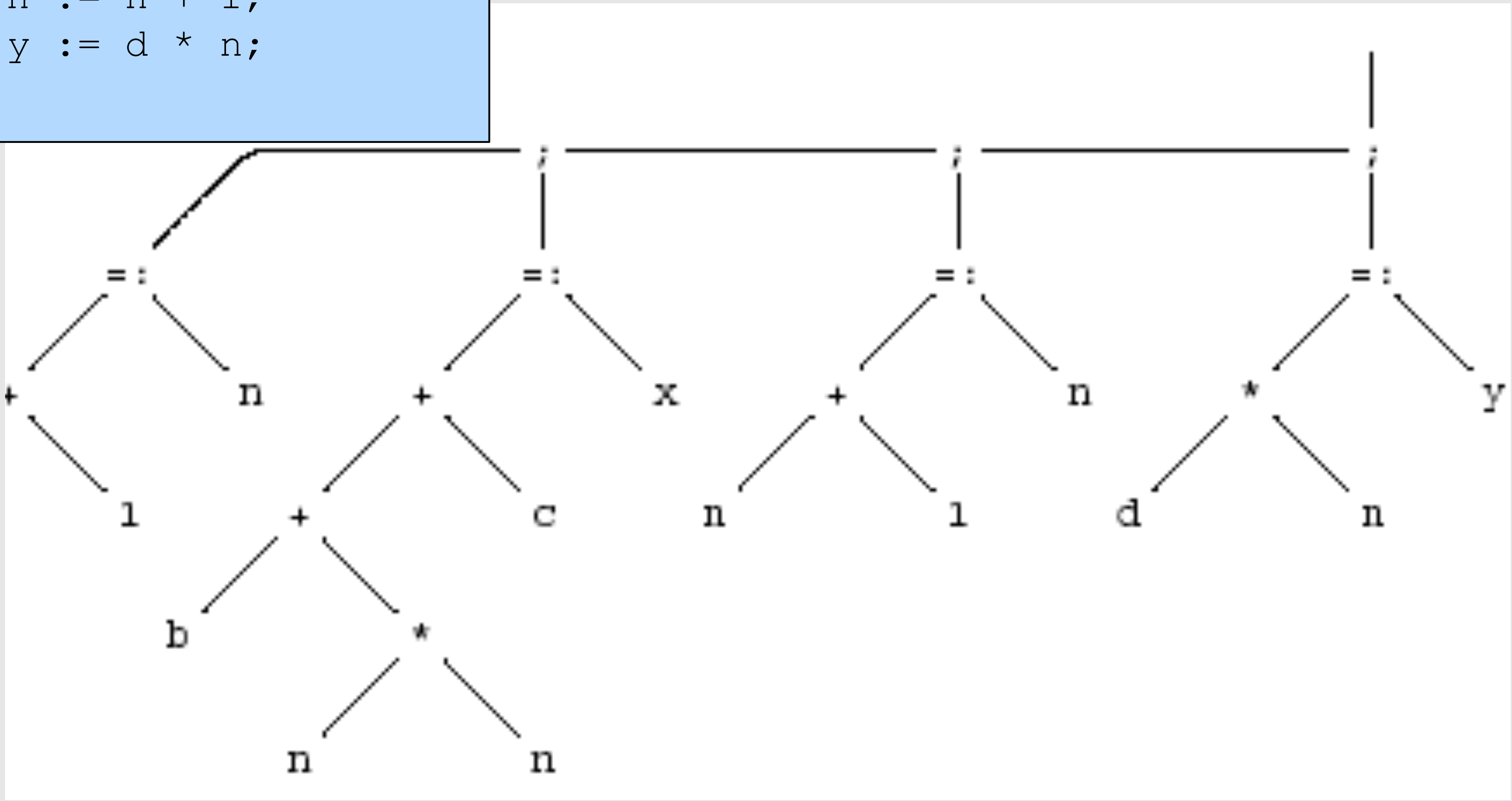
## _t0 = **cgen**( a+(b+(c*d)) )

*+ and * are commutative operators*



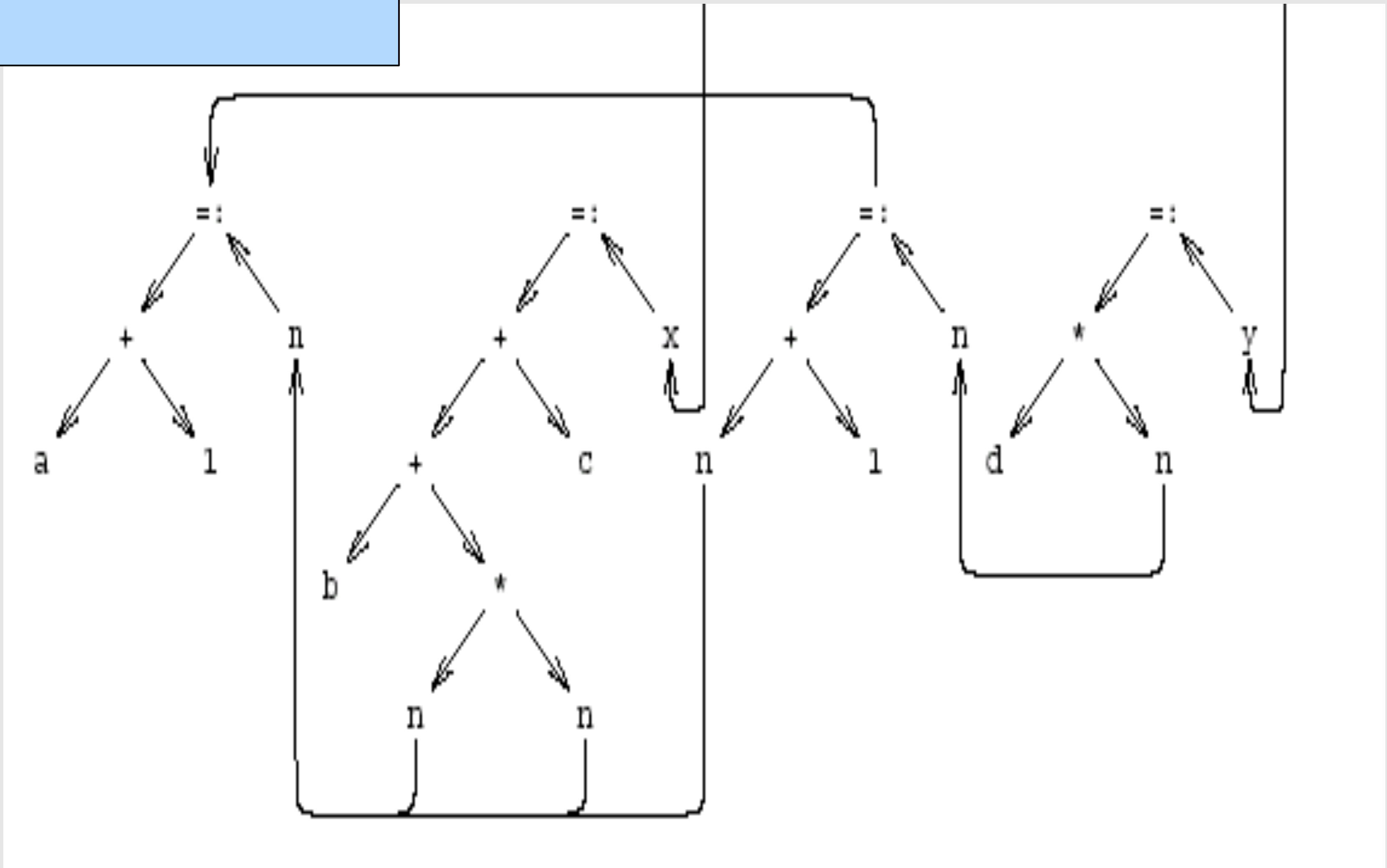
# AST for a Basic Block

```
{
    int n;
    n := a + 1;
    x := b + n * n + c;
    n := n + 1;
    y := d * n;
}
```



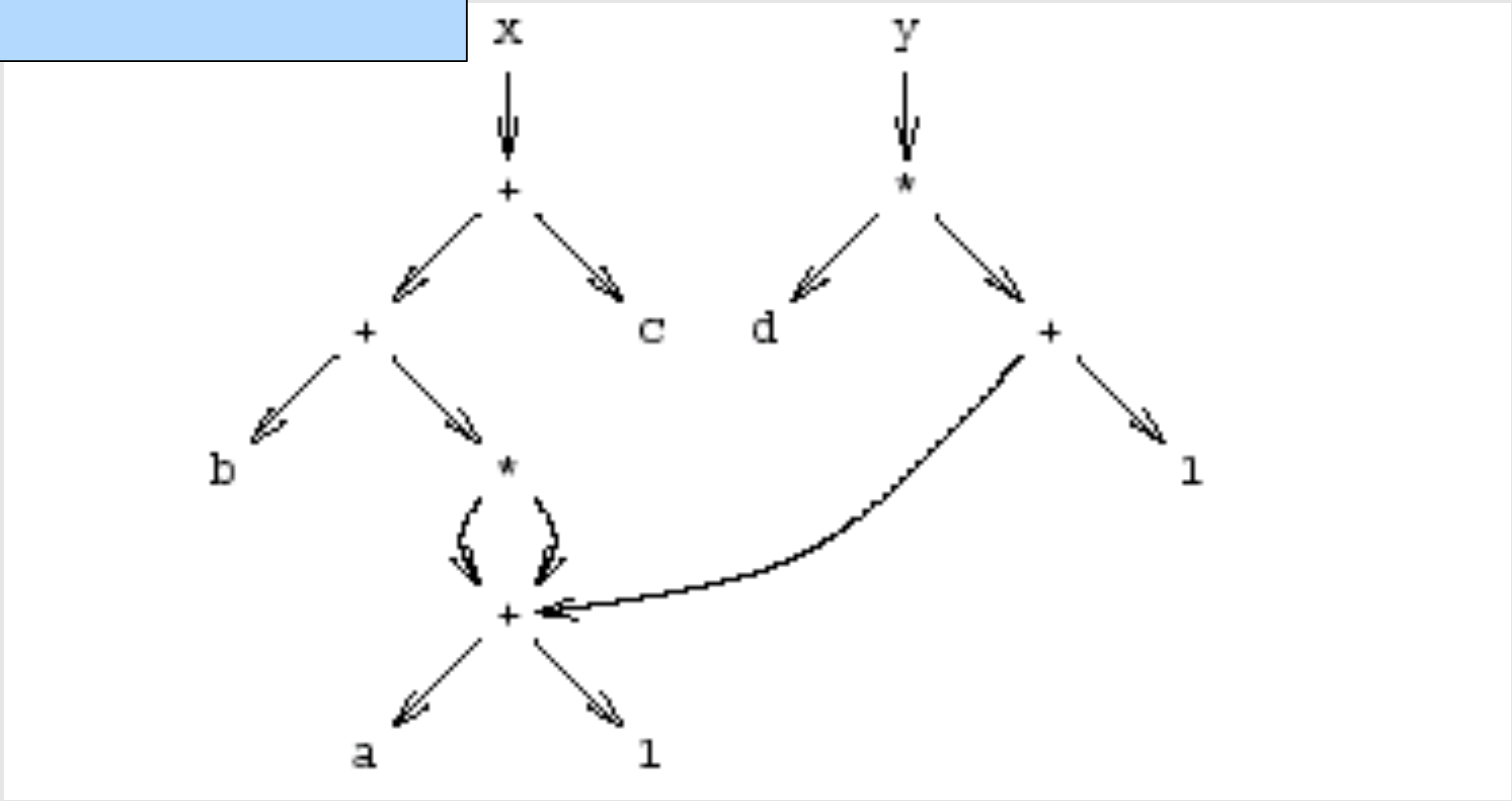
# Dependency graph

```
{
    int n;
    n := a + 1;
    x := b + n * n + c;
    n := n + 1;
    y := d * n;
}
```

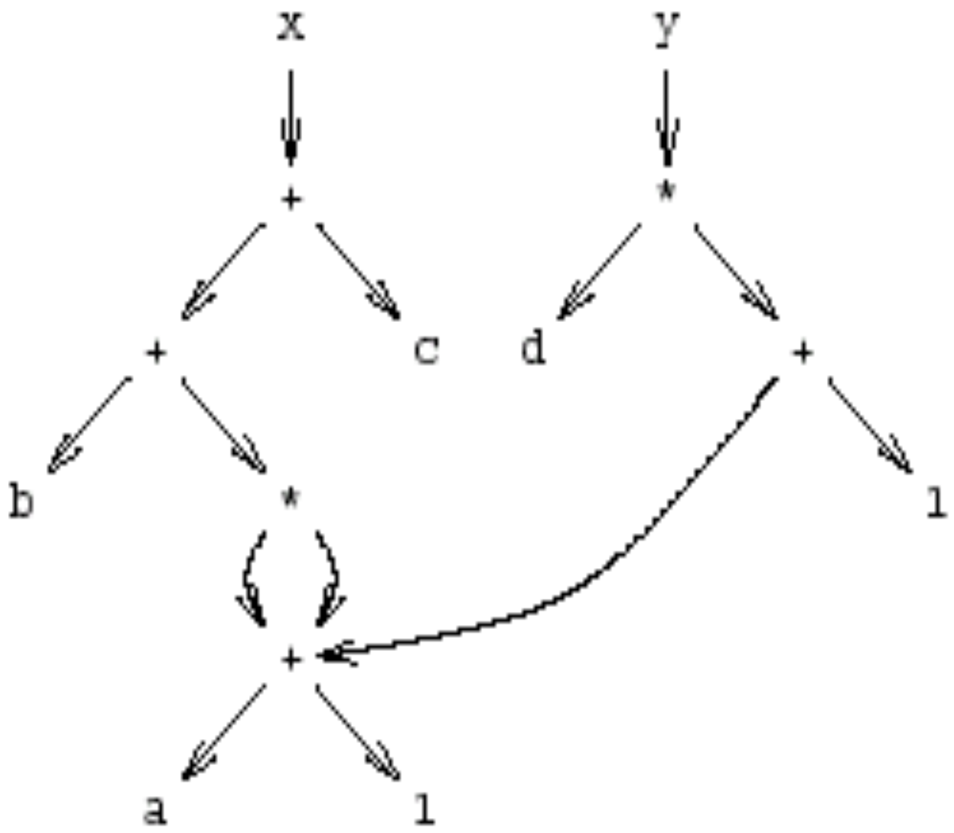


```
{
    int n;
    n := a + 1;
    x := b + n * n + c;
    n := n + 1;
    y := d * n;
}
```

# Simplified Data Dependency Graph



# Pseudo Register Target Code




```
Load_Mem     a,R1
Add_Const    1,R1
Load_Reg     R1,X1

Load_Reg     X1,R1
Mult_Reg     X1,R1
Add_Mem      b,R1
Add_Mem      c,R1
Store_Reg    R1,x

Load_Reg     X1,R1
Add_Const    1,R1
Mult_Mem     d,R1
Store_Reg    R1,y
```

# “Global” Register Allocation

- Input:
  - Sequence of machine instructions (“assembly”)
    - Unbounded number of temporary variables
      - aka symbolic registers
  - “machine description”
    - # of registers, restrictions
- Output
  - Sequence of machine instructions using machine registers (assembly)
  - Some MOV instructions removed

# Variable Liveness

- A statement x = y + z
  - **defines** x
  - **uses** y and z
- A variable x is live at a program point if its value (at this point) is used at a later point

```
y = 42
z = 73
x = y + z
print(x);
```

x undef, y live, z undef
x undef, y live, z live
x is live, y dead, z dead
x is dead, y dead, z dead

(showing state after the statement)

# Main idea

- For every node n in CFG, we have out[n]
  - Set of temporaries live out of n
- Two variables *interfere* if they appear in the same out[n] of any node n
  - **Cannot be allocated to the same register**
- Conversely, if two variables do not interfere with each other, they can be assigned the same register
  - We say they have disjoint live ranges
- How to assign registers to variables?

# Interference graph

- Nodes of the graph = variables
- Edges connect variables that interfere with one another
- Nodes will be assigned a color corresponding to the register assigned to the variable
- Two colors can't be next to one another in the graph

# Graph coloring

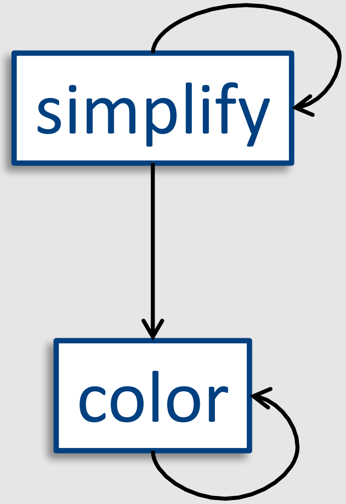
- This problem is equivalent to graph-coloring, which is NP-hard if there are at least three registers
- No good polynomial-time algorithms (or even good approximations!) are known for this problem
  - We have to be content with a heuristic that is good enough for RIGs that arise in practice

# Coloring by simplification [Kempe 1879]

- How to find a **k**-coloring of a graph
- Intuition:
  - Suppose we are trying to ***k****-color a graph and find a node* with fewer than ***k*** *edges*
  - If we delete this node from the graph and color what remains, we can find a color for this node if we add it back in
  - Reason: fewer than *k neighbors  some color must be* left over

# Coloring by simplification [Kempe 1879]

- How to find a k-coloring of a graph
- Phase 1: Simplification
  - Repeatedly simplify graph
  - When a variable (i.e., graph node) is removed, push it on a stack
- Phase 2: Coloring
  - Unwind stack and reconstruct the graph as follows:
  - Pop variable from the stack
  - Add it back to the graph
  - Color the node for that variable with a color that it doesn't interfere with



# Handling precolored nodes

- Some variables are pre-assigned to registers
  - Eg: mul on x86/pentium
    - uses eax; defines eax, edx
  - Eg: call on x86/pentium
    - Defines (trashes) caller-save registers eax, ecx, edx
- To properly allocate registers, treat these register uses as special temporary variables and enter into interference graph as precolored nodes

# Optimizing move instructions

- Code generation produces a lot of extra mov instructions

  mov t5, t9

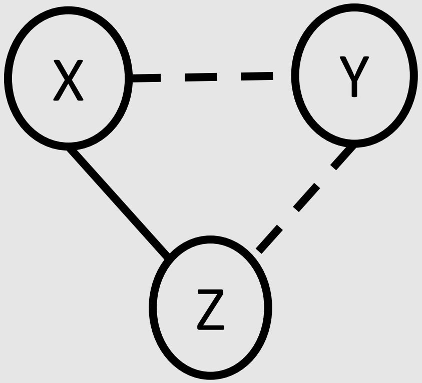
- If we can assign t5 and t9 to same register, we can get rid of the mov
  - effectively, copy elimination at the register allocation level
- Idea: if t5 and t9 are not connected in inference graph, coalesce them into a single variable; the move will be redundant
- Problem: coalescing nodes can make a graph un-colorable
  - Conservative coalescing heuristic

# Constrained Moves

- A instruction T ← S is constrained
  - if S and T interfere
- May happen after coalescing

X ← Y

Y ← Z



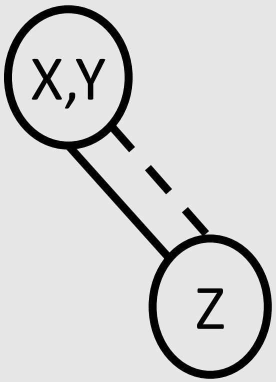

- Constrained MOVs are not coalesced

# Constrained Moves

- A instruction T ← S is constrained
  - if S and T interfere
- May happen after coalescing

X ← Y

Y ← Z



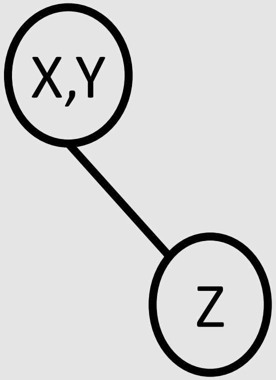

- Constrained MOVs are not coalesced

# Constrained Moves

- A instruction T ← S is constrained
  - if S and T interfere
- May happen after coalescing

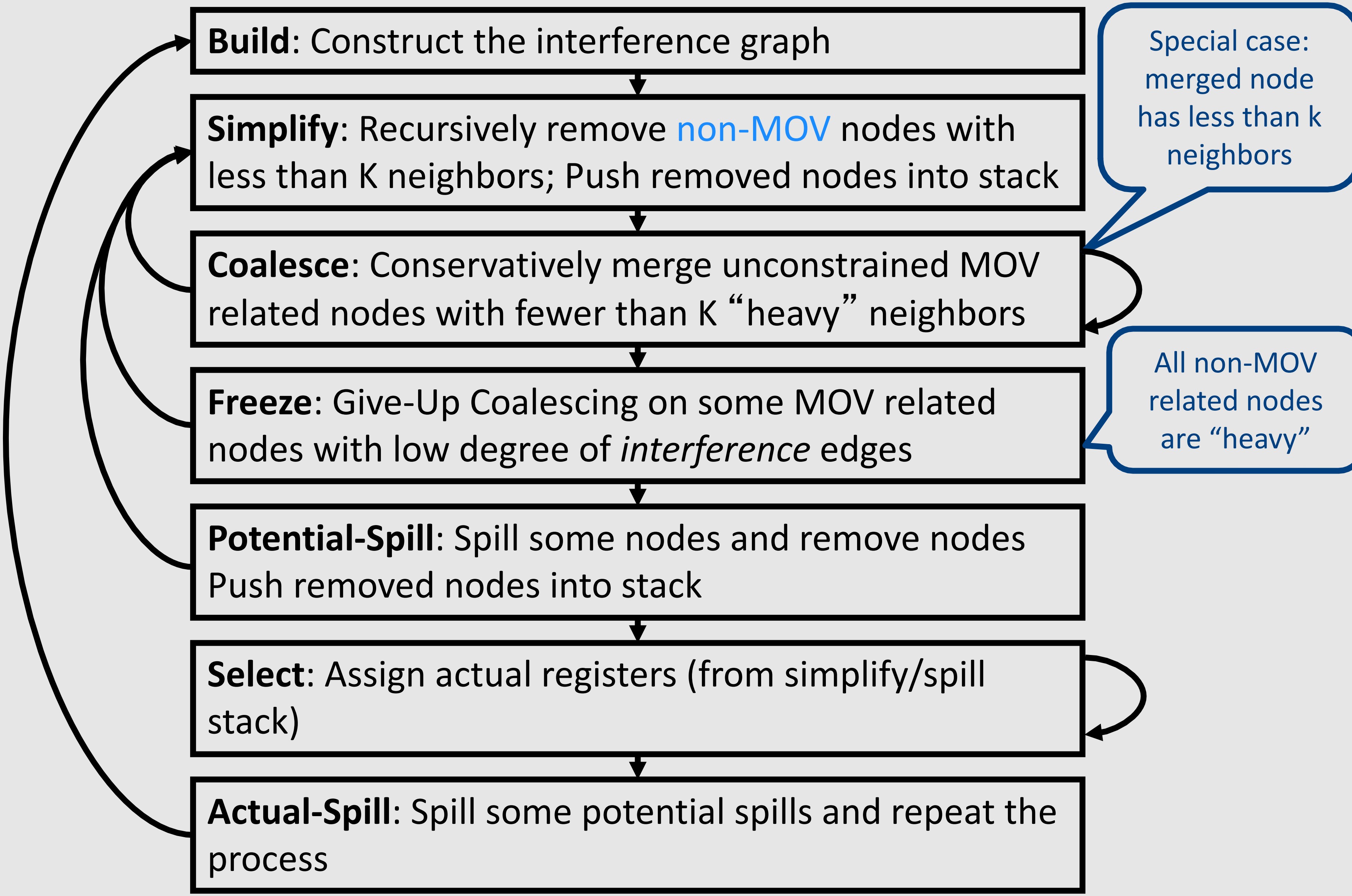
X ← Y

Y ← Z




- Constrained MOVs are not coalesced

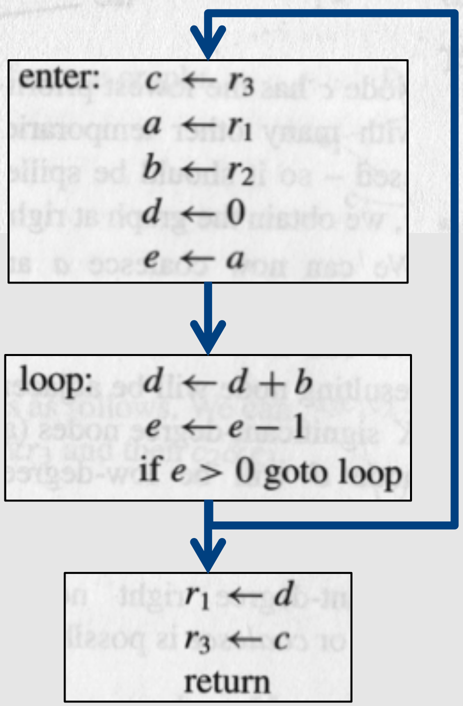
# Graph Coloring with Coalescing

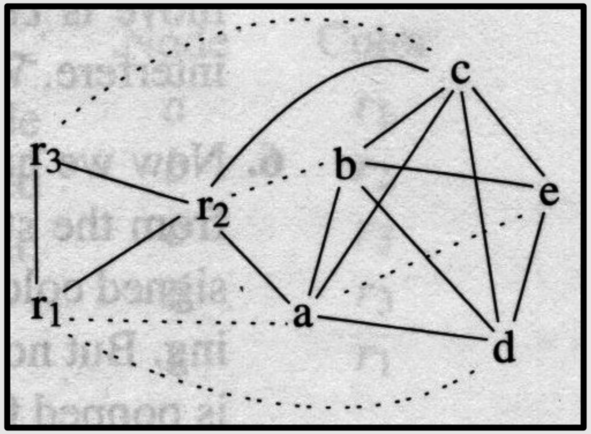


# A Complete Example

```
int f(int a, int b) {
   int d=0;
   int e=a;
   do {d = d+b;
       e = e-1;
   } while (e>0);
   return d;
}
```

enter: $c \leftarrow r_3$
$a \leftarrow r_1$ — Callee-saved registers
$b \leftarrow r_2$ — Caller-saved registers
$d \leftarrow 0$
$e \leftarrow a$
loop: $d \leftarrow d + b$
$e \leftarrow e - 1$
if $e > 0$ goto loop
$r_1 \leftarrow d$
$r_3 \leftarrow c$
return ($r_1, r_3$ *live out*)



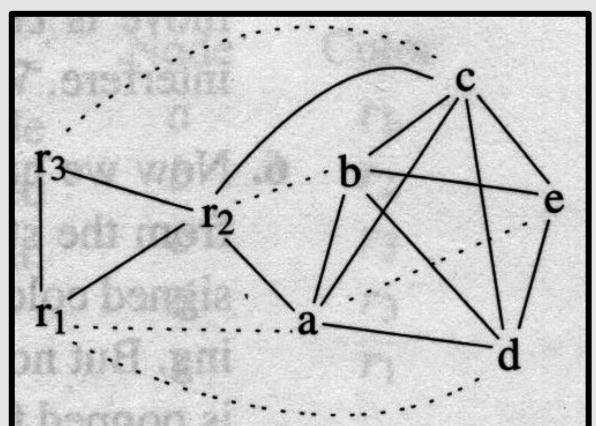



# A Complete Example

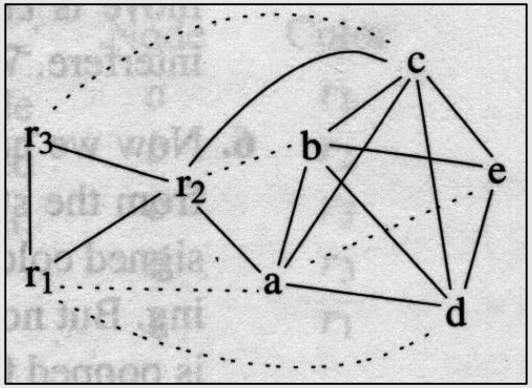
```
int f(int a, int b) {
   int d=0;
   int e=a;
   do {d = d+b;
       e = e-1;
   } while (e>0);
   return d;
}
```

enter: $c \leftarrow r_3$
$a \leftarrow r_1$
$b \leftarrow r_2$
$d \leftarrow 0$
$e \leftarrow a$
loop: $d \leftarrow d + b$
$e \leftarrow e - 1$
if $e > 0$ goto loop
$r_1 \leftarrow d$
$r_3 \leftarrow c$
return ($r_1, r_3$ *live out*)

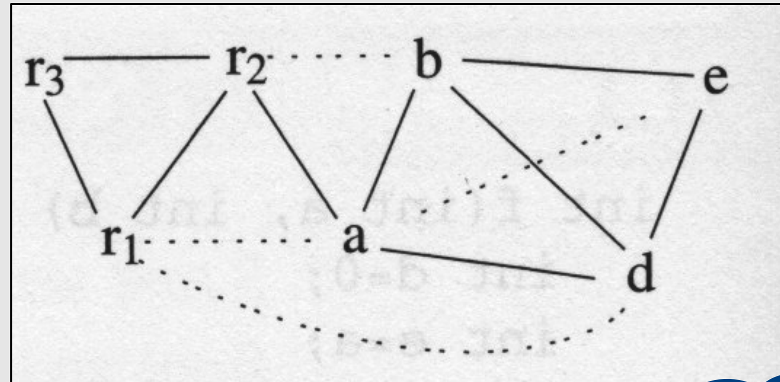
| Node | Uses+Defs outside loop | | Uses+Defs within loop | | | Degree | | Spill priority |
|---|---|---|---|---|---|---|---|---|
| *a* | ( 2 | + 10 × | 0 | ) | / | 4 | = | 0.50 |
| *b* | ( 1 | + 10 × | 1 | ) | / | 4 | = | 2.75 |
| *c* | ( 2 | + 10 × | 0 | ) | / | 6 | = | 0.33 |
| *d* | ( 2 | + 10 × | 2 | ) | / | 4 | = | 5.50 |
| *e* | ( 1 | + 10 × | 3 | ) | / | 3 | = | 10.33 |



# A Complete Example

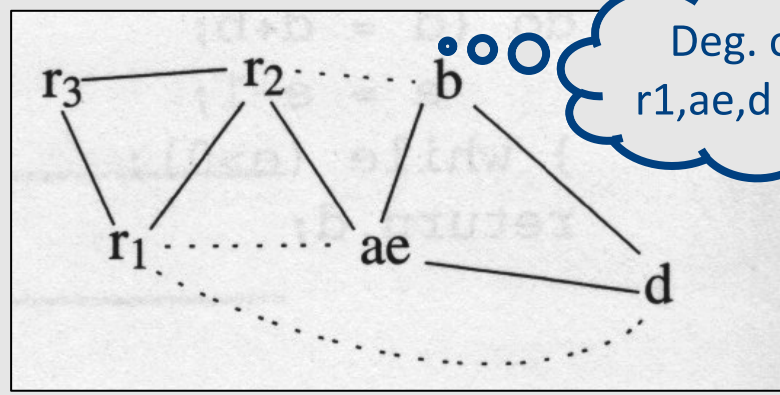



Spill c




c

a & e

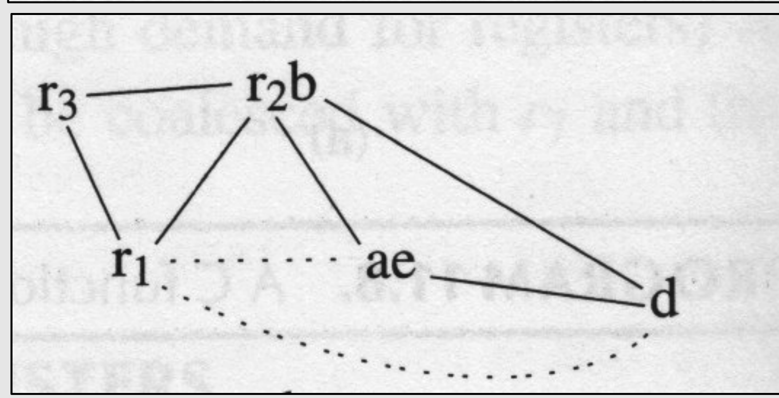



c

Deg. of
r1,ae,d < K

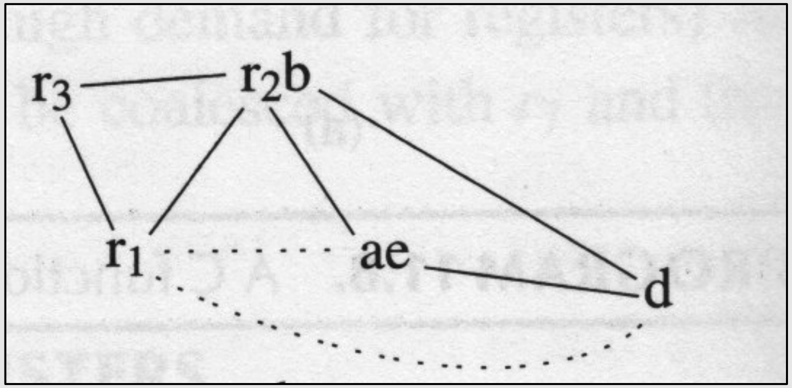
r2 & b

(Alt: ae+r1)



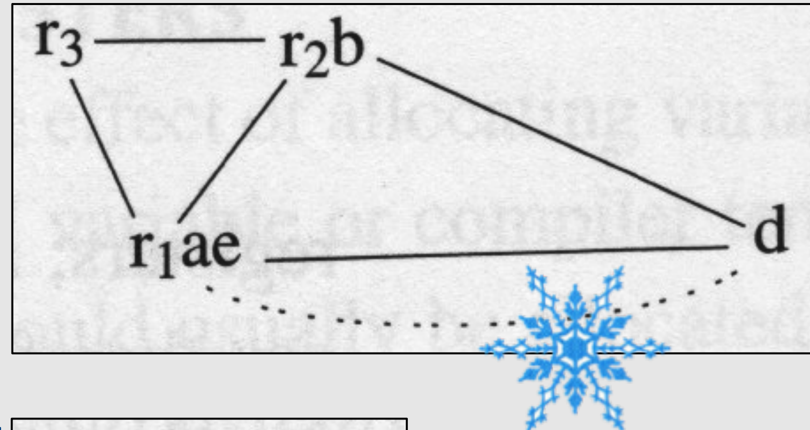

c

# A Complete Example



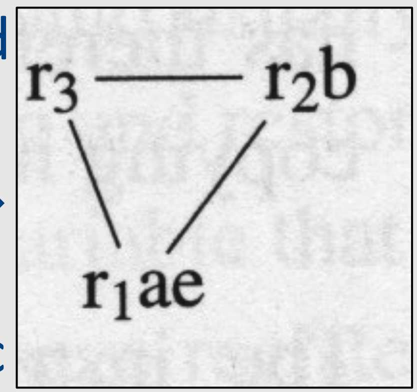

ae & r1

(Alt: …)c



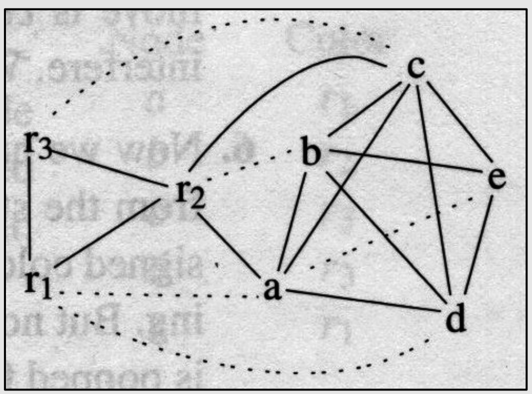

freeze $r_1$ae-d

Simplify d




dc

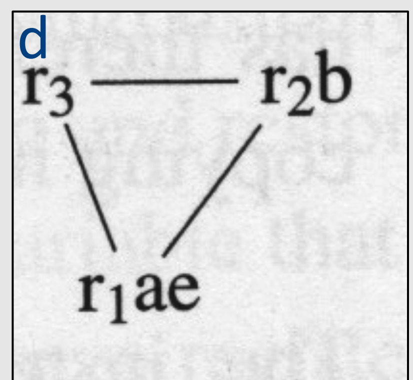
pop c …




(Alt: ae+r1)

pop d

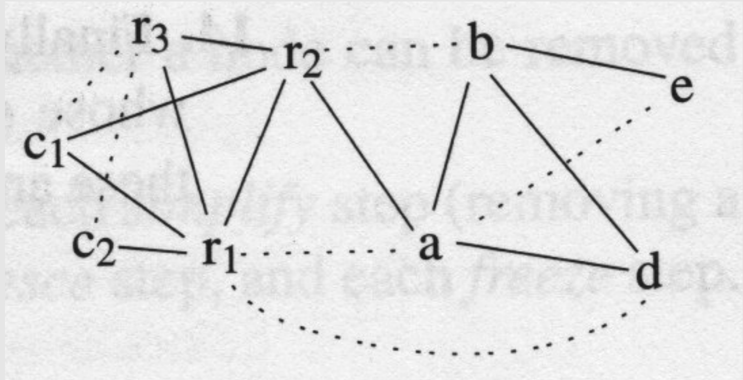
d



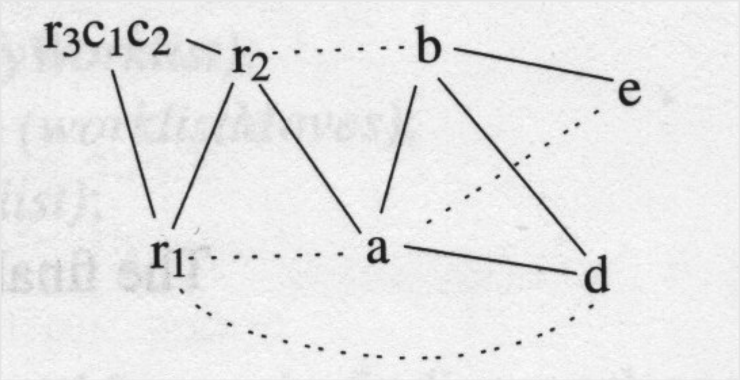

c

# A Complete Example

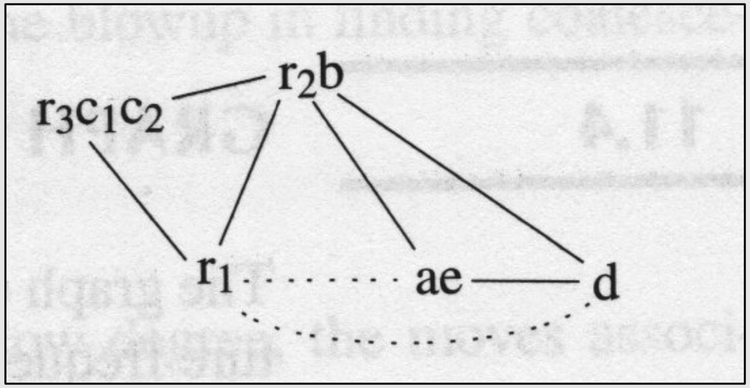
enter: $c_1 \leftarrow r_3$
$M[c_{loc}] \leftarrow c_1$
$a \leftarrow r_1$
$b \leftarrow r_2$
$d \leftarrow 0$
$e \leftarrow a$
loop: $d \leftarrow d + b$
$e \leftarrow e - 1$
if $e > 0$ goto loop
$r_1 \leftarrow d$
$c_2 \leftarrow M[c_{loc}]$
$r_3 \leftarrow c_2$
return



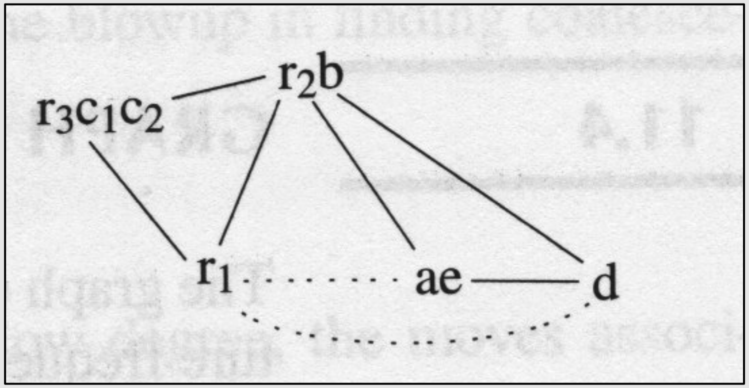

c1&r3, c2 &r3



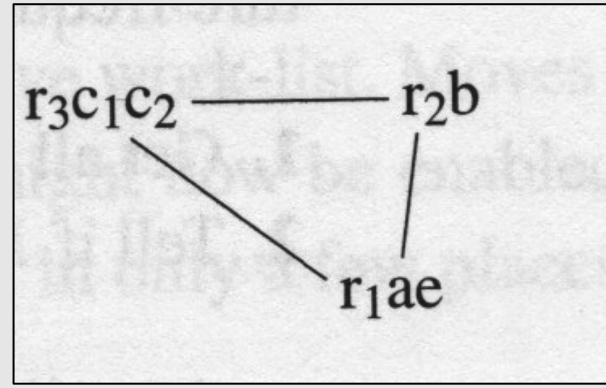

a&e, b&r2



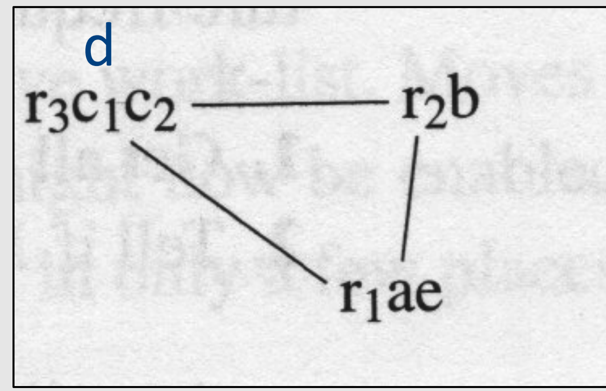
# A Complete Example




ae & r1
Simplify d




Pop d




gen code

```
enter:  r3 ← r3
        M[cloc] ← r3
        r1 ← r1
        r2 ← r2
        r3 ← 0
        r1 ← r1
loop:   r3 ← r3 + r2
        r1 ← r1 − 1
        if r1 > 0 goto loop
        r1 ← r3
        r3 ← M[cloc]
        r3 ← r3
        return
```

"opt"

```
enter:  M[cloc] ← r3
        r3 ← 0
loop:   r3 ← r3 + r2
        r1 ← r1 − 1
        if r1 > 0 goto loop
        r1 ← r3
        r3 ← M[cloc]
        return
```

# Compiling OO Programs

# Features of OO languages

- **Inheritance**
  - Subclass gets (inherits) properties of superclass
- **Method overriding**
  - Multiple methods with the same name with different signatures
- **Abstract** (aka **virtual**) **methods**
- **Polymorphism**
  - Multiple methods with the same name and different signatures but with different

# Compiling OO languages

- “Translation into C”
- Powerful runtime environment

- Adding “gluing” code

# Runtime Environment

- Mediates between the OS and the programming language
- Hides details of the machine from the programmer
  - Ranges from simple support functions all the way to a full-fledged virtual machine
- Handles common tasks
  - Runtime stack (activation records)
  - Memory management

# Handling Single Inheritance

- Simple type extension

```
class A {
    field a1;
    field a2;
    method m1() {…}
    method m2() {…}
}
```

```
class B extends A {
    field b1;
    method m3() {…}
}
```

# Adding fields

Fields aka Data members, instance variables

- Adds more information to the inherited

```
class A {
    field a1;
    field a2;
    method m1() {…}
    method m2() {…}
}
```

s ensures

```
class B extends A {
    field b1;
    method m2() {…}
    method m3() {…}
}
```

```
typedef struct  {
    field a1;
    field a2;
} A;

void m1A_A(A* this){…}
void m2A_A(A* this){…}
```

```
typedef struct  {
    field a1;
    field a2;
    field b1;
} B;

void m2A_B(B* this) {…}
void m3B_B(B* this) {…}
```

# Method Overriding

- Redefines functionality
  - More specific
  - Can access additional fields

```
class A {
    field a1;
    field a2;
    method m1() {…}
    method m2() {…}
}
```

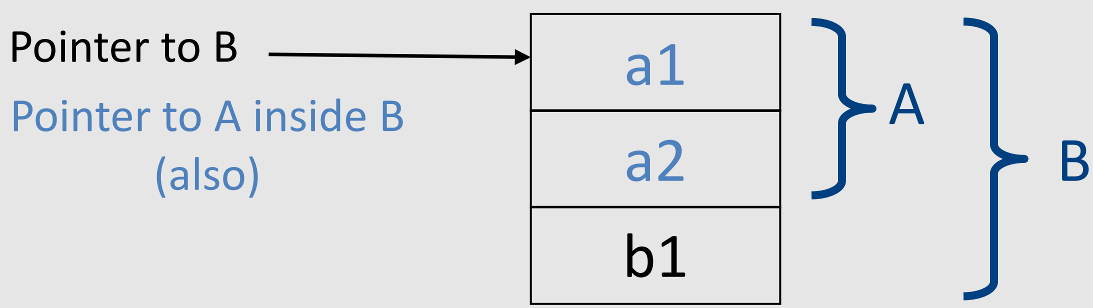
```
class B extends A {
    field b1;
    method m2() {
        … b1 …
    }
    method m3() {…}
}
```

# Handling Polymorphism

- When a class B extends a class A
  - variable of type pointer to A may actually refer to object of type B
- Upcasting from a subclass to a superclass
- Pr

```
class B *b = …;
class A *a = b ;
```

```
classA *a = convert_ptr_to_B_to_ptr_A(b) ;
```



# Dynamic Binding

- An object (“pointer”) o declared to be of class A can actually be (“refer”) to a class B
- What does ‘o.m()’ mean?
  - Static binding
  - Dynamic binding
- Depends on the programming language rules

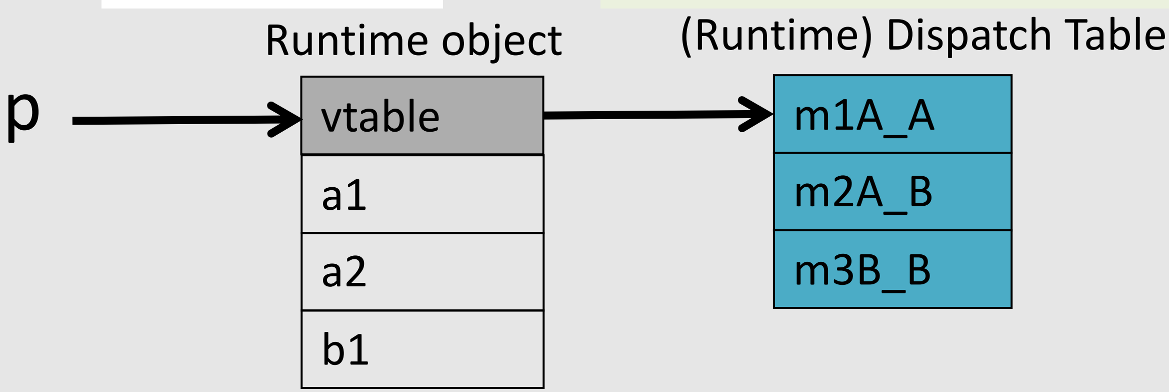
```
typedef struct  {
    field a1;
    field a2;
} A;

void m1A_A(A* this){…}
void m2A_A(A* this, int x){…}
```

```
typedef struct  {
    field a1;
    field a2;
    field b1;
} B;

void m2A_B(A* thisA, int x){
   Class_B *this =
    convert_ptr_to_A_to_ptr_to_B(thisA);
    …
  }

void m3B_B(B* this){…}
```

p.m2(3);

convert_ptr_to_B_to_ptr_to_A(p)

p→dispatch_table→m2A( , 3);



# Multiple Inheritance

```
class C {
    field c1;
    field c2;
    method m1(){…}
    method m2(){…}
}
```

```
class D {
    field d1;

    method m3() {…}
    method m4(){…}
}
```

```
class E extends C, D {
    field e1;

    method m2() {…}
    method m4() {…}
    method m5(){…}
}
```

# Multiple Inheritance

- Allows unifying behaviors
- But raises semantic difficulties
  - Ambiguity of classes
  - Repeated inheritance
- Hard to implement
  - Semantic analysis
  - Code generation
    - Prefixing no longer work
    - Need to generate code for downcasts

# A simple implementation

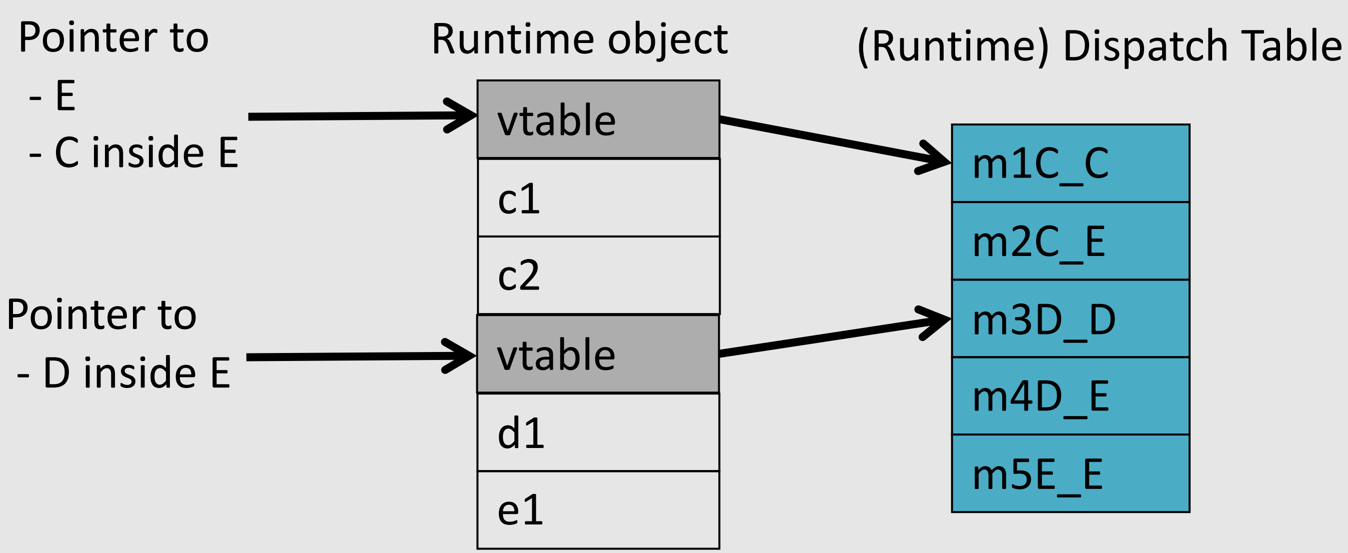
- Merge dispatch tables of superclases
- Generate code for upcasts and downcasts

# A simple implementation

```
class C {
    field c1;
    field c2;
    method m1(){…}
    method m2(){…}
}
```

```
class D {
    field d1;

    method m3() {…}
    method m4(){…}
}
```

```
class E extends C, D {
    field e1;

    method m2() {…}
    method m4() {…}
    method m5(){…}
}
```



# Dependent multiple Inheritance

```
class A {
    field a1;
    field a2;
    method m1(){…}
    method m3(){…}
}
```

```
class C extends A {
    field c1;
    field c2;
    method m1(){…}
    method m2(){…}
}
```

```
class D extends A {
    field d1;

    method m3(){…}
    method m4(){…}
}
```

```
class E extends C, D {
    field e1;

    method m2() {…}
    method m4() {…}
    method m5(){…}
}
```

# Interface Types

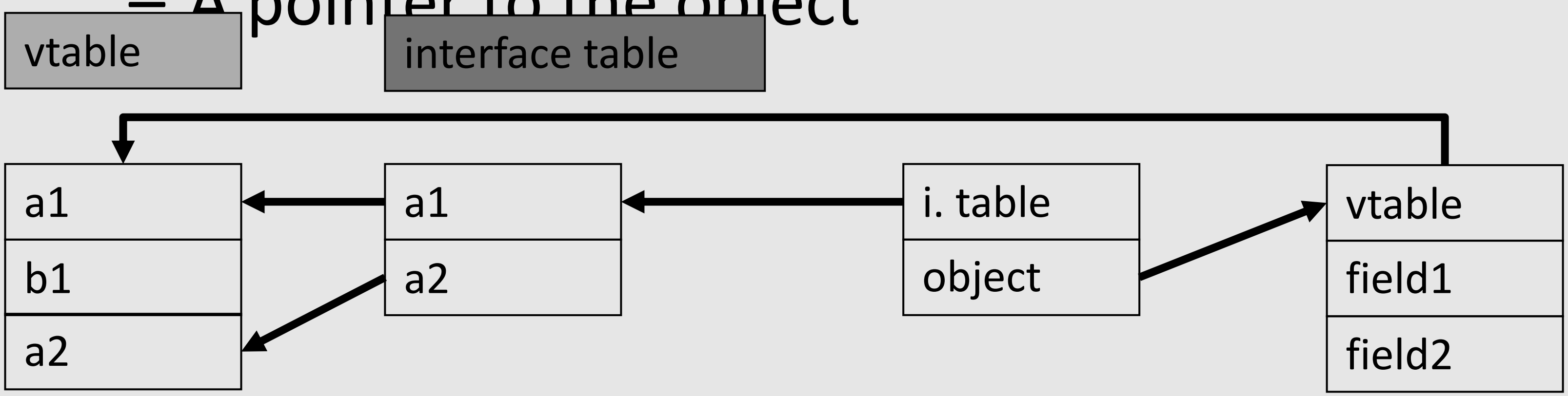
- Java supports limited form of multiple inheritance
- Interface consists of several methods, but no fields

```
public interface Comparable {
    public int compareTo(Comparable o);
}
```

- A class can implement multiple interfaces

# Interface Types

- Implementation: record with 2 pointers:
  - A separate dispatch table per interface
  - A pointer to the object



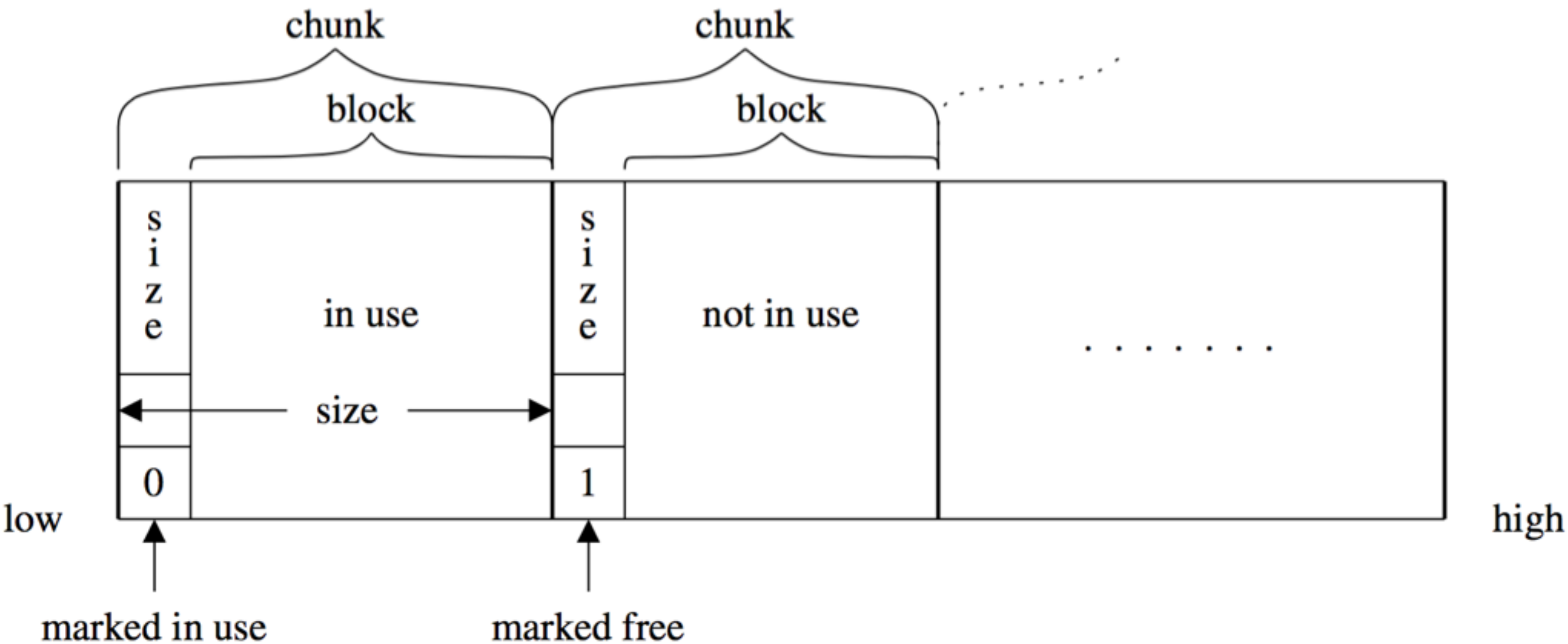
# Memory Management

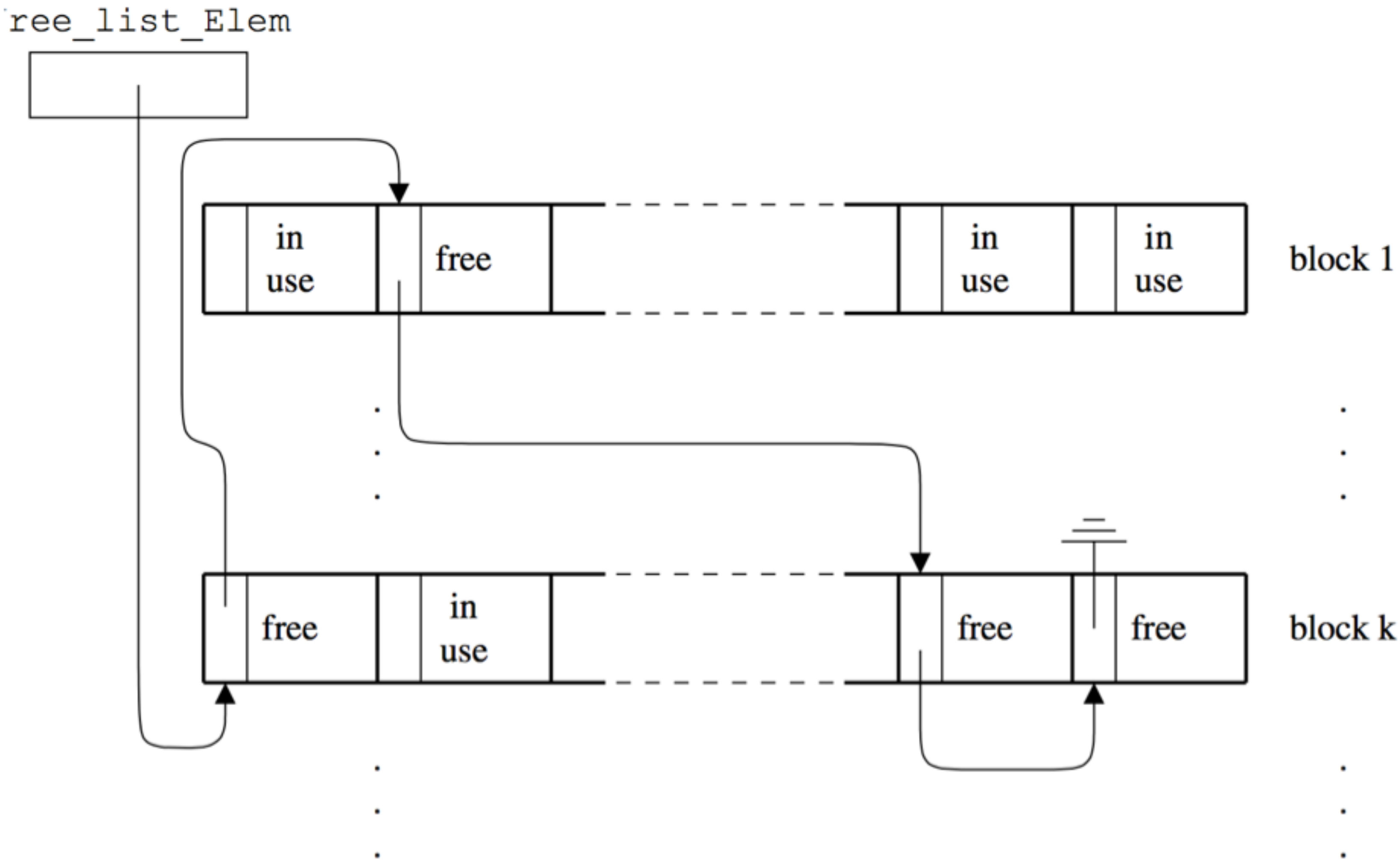
- Manual memory management
- Automatic memory management

# Free-list Allocation

- A data structure records the location and size of free cells of memory.
- The allocator considers each free cell in turn, and according to some policy, chooses one to allocate.
- Three basic types of free-list allocation:
  - First-fit
  - Next-fit
  - Best-fit

# Memory chunks




'ree_list_Elem
in
use
free
in
use
in
use
block 1
free
in
use
free
free
block k

# free

- Free too late – waste memory (memory leak)
- Free too early – dangling pointers / crashes
- Free twice – error

# Garbage collection

- approximate reasoning about object liveness
- use reachability to approximate liveness
- **assume reachable objects are live**
  - non-reachable objects are dead

# Garbage Collection – Classical Techniques

- reference counting
- mark and sweep
- copying

# GC using Reference Counting

- add a reference-count field to every object
  - how many references point to it
- when (rc==0) the object is non reachable
  - non reachable => dead
  - can be collected (deallocated)

# The Mark-and-Sweep Algorithm [McCarthy 1960]

- Marking phase
  - mark roots
  - trace all objects transitively reachable from roots
  - mark every traversed object

- Sweep phase
  - scan all objects in the heap
  - collect all unmarked objects

# Mark&Sweep in Depth

```
mark(Obj)=
if mark_bit(Obj) == unmarked
        mark_bit(Obj)=marked
        for C in Children(Obj)
                mark(C)
```

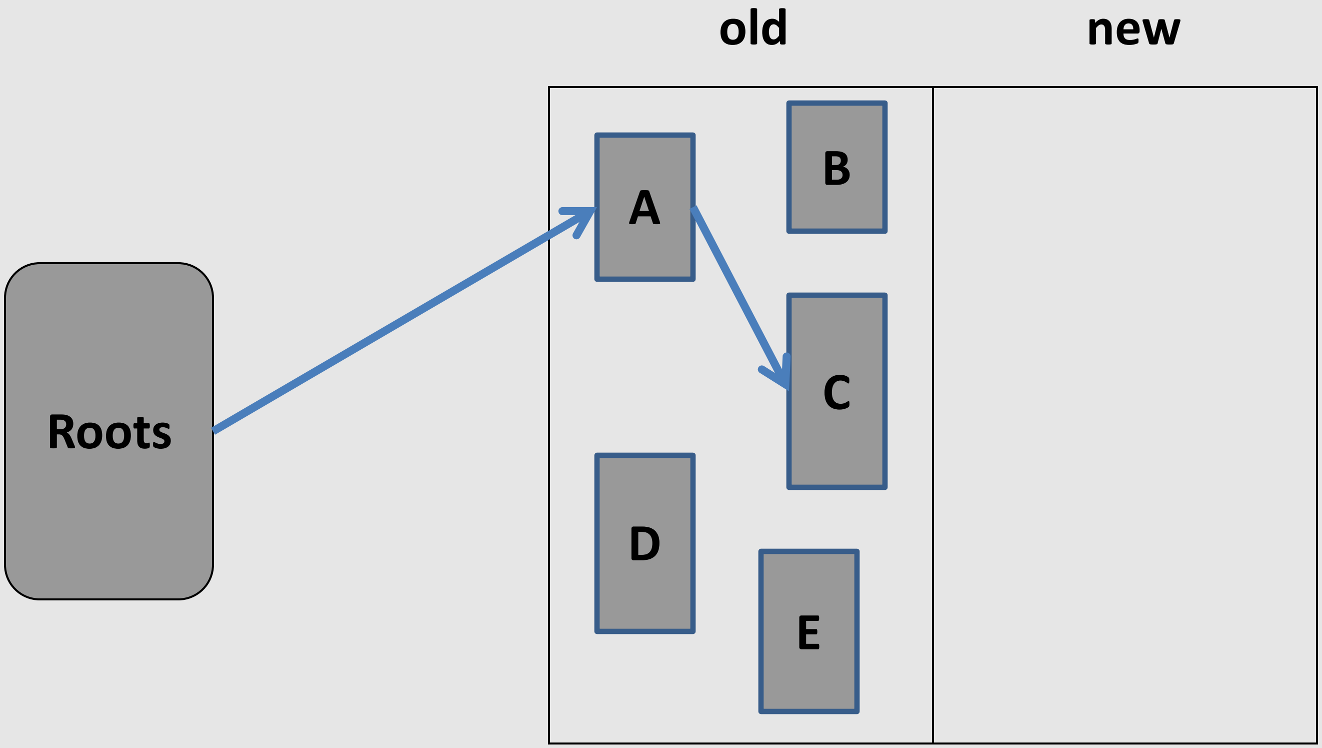
- How much memory does it consume?
  - Recursion depth?
  - Can you traverse the heap without worst-case O(n) stack?
    - Deutch-Schorr-Waite algorithm for graph marking without recursion or stack (works by reversing pointers)

# Copying GC

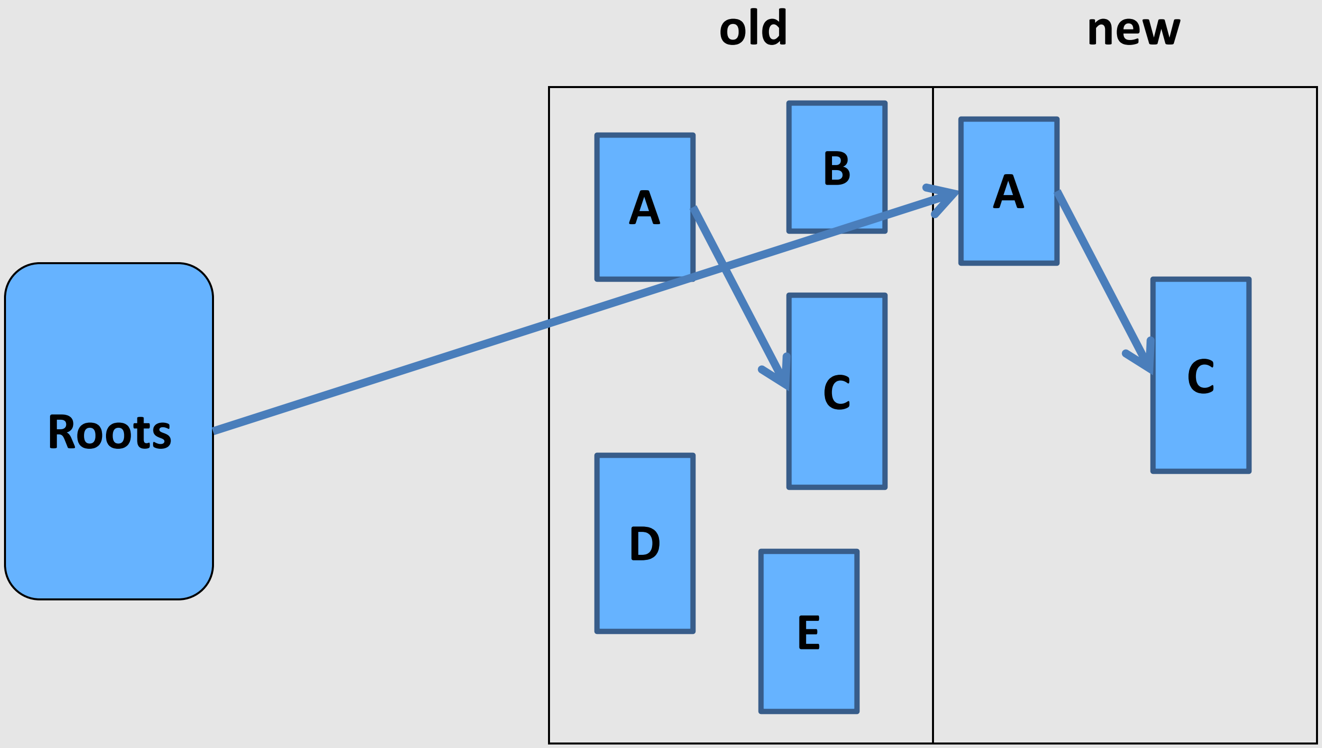
- partition the heap into two parts
  - old space
  - new space

- Copying GC algorithm
  - copy all **reachable** objects from old space to new space
  - swap roles of old/new space

# Example



# Example



# The Exam

מרצה: **נעם רינצקי** מתרגל: **אורן איש שלום** חומר: **פתוח** משך: **שלוש שעות**

**מבחן בקומפילציה – 2017/18 – מועד א**

- המבחן מורכב מחמש שאלות. **יש לענות על כולן**.
- מומלץ לקרוא את השאלה עד סופה לפני שמתחילים לענות.
- משקל השאלה ומספרה אינה מעיד על הקושי בפתירתה.
- **יש לציין בראש העמוד את השאלה עליה עונים.**
- **אין לענות על שאלות שונות באותו העמוד.**
- תשובה "איני יודע/ת" תזכה ב 20% מהניקוד על הסעיף הרלוונטי .